

Community Experience Distilled

Python Machine Learning

Unlock deeper insights into machine learning with this vital guide to cutting-edge predictive analytics

Foreword by Dr. Randal S. Olson
Artificial Intelligence and Machine Learning Researcher, University of Pennsylvania

Sebastian Raschka

[PACKT] open source*
PUBLISHING
community experience distilled

目錄

Python机器学习

机器学习，如今最令人振奋的计算机领域之一。看看那些大公司，Google、Facebook、Apple、Amazon早已展开了一场关于机器学习的军备竞赛。从手机上的语音助手、垃圾邮件过滤到逛淘宝时的物品推荐，无一不用到机器学习技术。

如果你对机器学习感兴趣，甚至是想从事相关职业，那么这本书非常适合作为你的第一本机器学习资料。市面上大部分的机器学习书籍要么是告诉你如何推导模型公式要么就是如何代码实现模型算法，这对于零基础的新手来说，阅读起来相当困难。而这本书，在介绍必要的基础概念后，着重从如何调用机器学习算法解决实际问题入手，一步一步带你入门。即使你已经对很多机器学习算法的理论很熟悉了，这本书仍能从实践方面带给你一些帮助。

具体到编程语言层面，本书选择的是Python，因为它简单易懂。我们不必在枯燥的语法细节上耗费时间，一旦有了想法，你能够快速实现算法并在真实数据集上进行验证。在整个数据科学领域，Python都可以说是稳坐语言榜头号交椅。

注意 这是我在阅读 [Python Machine Learning](https://www.amazon.com/Python-Machine-Learning-Sebastian-Raschka/dp/1783555130) 的笔记而不是严格的原文翻译，如果发现错误，请不惜吝教~

我的邮箱: ljalphabeta@gmail.com

第一章 让计算机从数据中学习

我个人认为，机器学习(machine learning)是计算机领域中最令人振奋的方向！我们生活在一个数据爆炸的时代，通过运用机器学习领域中的自学习算法，我们可以将数据转化成知识。感谢近年来这一领域中开源软件的蓬勃发展，这真是一个美好的时代。

这一章，我们将学习机器学习的一些重要概念和几类常用算法。最主要的是可以借助Python和开源机器学习库来解决实际的问题。

我们要讲到的主题包括：

- 机器学习的一些概念
- 三类机器学习算法
- 如何设计一个成熟的机器学习系统
- 安装Python

建立智能机器将数据转化为知识

如今，我们最不缺的就是数据：不论是结构化数据还是非结构数据。在20世纪下半叶，机器学习作为人工智能(artificial intelligence)的子领域诞生了，她的目标是通过自学习算法从数据中获取知识，然后对未来世界进行预测。无需借助人力从数据中得到规则，机器学习能够自动建立模型进行预测。机器学习不但在计算机学科中变得越发重要，她还在我们的日常生活中发挥了很大作用。有了机器学习，我们才能享受到垃圾邮件过滤技术、文字和语音识别技术、可信赖的网页搜索技术和自动驾驶汽车。

三类机器学习算法

第二章 训练机器学习分类算法

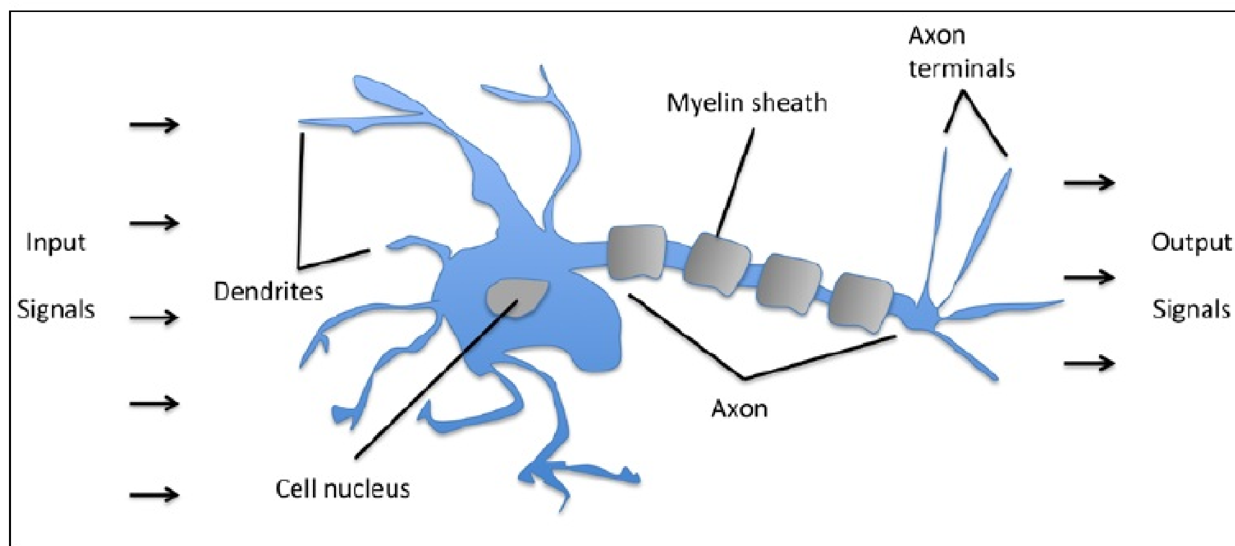
在本章，我们要学习第一个有确定性算法描述的机器学习分类算法，感知机(perceptron)和自适应线性神经元(adaptive linear neurons)。 我们首先用Python实现一个感知机算法，然后将其应用于Iris数据集。通过代码实现会帮助我们更好地理解分类的概念以及如果用Python有效地实现算法。 学习自适应线性神经元算法涉及到的优化知识，为我们第三章运用scikit-learn中更加复杂的分类器打好数学基础。

本章涉及到的知识点包括：

- 对机器学习算法有直观了解
- 使用pandas, NumPy和matplotlib读入数据，处理数据和可视化数据
- 使用Python实现线性分类算法

透过人工神经元一窥早期机器学习历史

在我们讨论感知机及其相关算法细节前，先让我们回顾一下机器学习早期的发展历程。为了理解大脑工作原理进而设计人工智能，Warren McCullock和Walter Pitts 在1943年首次提出了一个简化版大脑细胞的概念，即McCullock-Pitts(MCP)神经元(W.S.McCulloch and W.Pitts. *A Logical Calculus of the Ideas Immanent in Nervous Activity.*)。神经元是大脑中内部连接的神经细胞，作用是处理和传播 化学和电信号，可见下图：



McCullock和Pitts描述了如下的神经细胞：可以看做带有两个输出的简单逻辑门；即有多个输入传递到树突，然后在神经元内部进行输入整合，如果累积的信号量超过某个阈值，会产生一个输出信号并且通过轴突进行传递。 十几年后，基于MCP神经元模型，Frank Rosenblatt 发表了第一个感知机学习规则(F.Rosenblatt, *The Perceptron, a Perceiving and Recognizing Automaton.* Cornell Aeronautical Laboratory, 1957)。 基于此感知机规则，Rosenblatt提出了能够自动学习最优权重参数的算法，权重即输入特征的系数。在监督学习和分类任务语境中，上面提到的算法还能够用于预测一个样本是属于类别A还是类别B。

更准确的描述是，我们可以将上面提到的样本属于哪一个类别这个问题称之为二分类问题(binary classification task),我们将其中涉及到的两个类别记作1(表示正类)和-1(表示负类)。我们再定义一个称为激活函数(activation function)的东东，激活函数接收一个输入向量x和相应的权重向量w的线性组合，其中z也被称为网络输入($z = w_1x_1 + \ldots + w_mx_m$):

$$\boldsymbol{w} = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}, \quad \boldsymbol{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}$$

此时，如果某个样本$x^{(i)}$的激活值，即 大于事先设置的阈值 ,我们就说样本$x^{(i)}$属于类别1，否则属于类别-1。

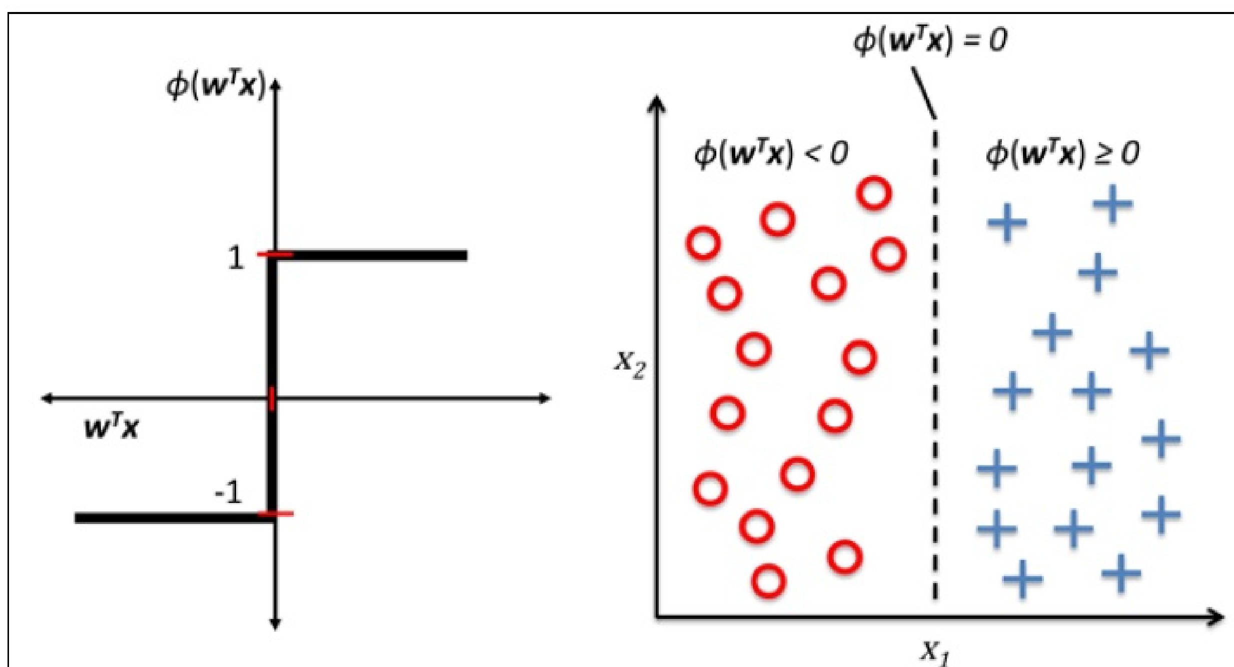
在感知机学习算法中，激活函数 的形式非常简单，仅仅是一个单位阶跃函数(也被称为Heaviside阶跃函数):

$$\phi(z)=\begin{cases} 1 & if\ z \geq \theta \\ -1 & otherwise \end{cases}$$

为了推导简单，我们可以将阈值 挪到等式左边并且额外定义一个权重参数 , 这样我们可以对z给出更加紧凑的公式$z = w_0x_0 + w_1x_1 + \ldots + w_mx_m = w^Tx$，此时

$$\phi(z)=\begin{cases} 1 & if\ z \geq 0 \\ -1 & otherwise \end{cases}$$

下面左图描述了感知机的激活函数怎样将网络输入$z = w^Tx$压缩到二元输出(-1,1)，右图描述了感知机如何区分两个线性可分的类别。



不论MCP神经元还是Rosenblatt的阈值感知机模型，他们背后的idea都是试图使用简单的方法来模拟大脑中单个神经元的工作方式：要么传递信号要么不传递。因此，Rosenblatt最初的感知机规则非常简单，步骤如下：

-
  1. 将权重参数初始化为0或者很小的随机数。
-
  1. 对于每一个训练集样本$x^{(i)}$,执行下面的步骤：

-
    - 1、计数输出值.
-
    - 2、更新权重参数.

此处的输出值就是单位阶跃函数预测的类别(1,-1)，参数向量w中的每个w_j的更新过程可以用数学语言表示为：

$$w_j := w_j + \Delta w_j$$.

其中，用于更新权重w_j,在感知机算法中的计算公式为:

$$\Delta w_j = \eta\left(y^{(i)} - \hat{y}^{(i)}\right)x_j^{(i)}$$

其中 称为学习率(learning rate), 是一个介于0.0和1.0之间的常数，$y^{(i)}$是第i个训练样本的真实类别，是对第i个训练样本的预测类别。 权重向量中的每一个参数**w_{j}**是同时被更新的，这意味着在所有的计算出来以前不会重新计算 (译者注：通俗地说，我们在计算出一个，然后同时更新w中的每一个权重参数；然后不断重复上面的步骤)。具体地，对于一个二维数据集，我们可以将更新过程写为：

$$\Delta w_0 = \eta\left(y^{(i)} - output^{(i)}\right)$$

$$\Delta w_1 = \eta\left(y^{(i)} - output^{(i)}\right)x_1^{(i)}$$

$$\Delta w_2 = \eta\left(y^{(i)} - output^{(i)}\right)x_2^{(i)}$$

在我们用Python实现感知机算法之前，我们先来草算一下这个简单的算法是多么美妙。如果感知机预测的类别正确，则权重参数不做改变，因为：

$$\Delta w_j = \eta\left(-1^{(i)} - -1^{(i)}\right)x_j^{(i)} = 0$$

$$\Delta w_j = \eta\left(1^{(i)} - 1^{(i)}\right)x_j^{(i)} = 0$$

当预测结果不正确时，权重会朝着正确类别方向更新(译者注：如果正确类别是1，权重参数会增大；如果正确类别是-1，权重参数会减小):

$$\Delta w_j = \eta\left(1^{(i)} - -1^{(i)}\right)x_j^{(i)} = \eta(2)x_j^{(i)}$$

$$\Delta w_j = \eta\left(-1^{(i)} - 1^{(i)}\right)x_j^{(i)} = \eta(-2)x_j^{(i)}$$

我们可以具体化上面的乘数$x_j^{(i)}$,比如一个简单的例子：

$$\hat{y}_j^{(i)} = +1,\ \ y^{(i)} = -1,\ \ \eta = 1$$

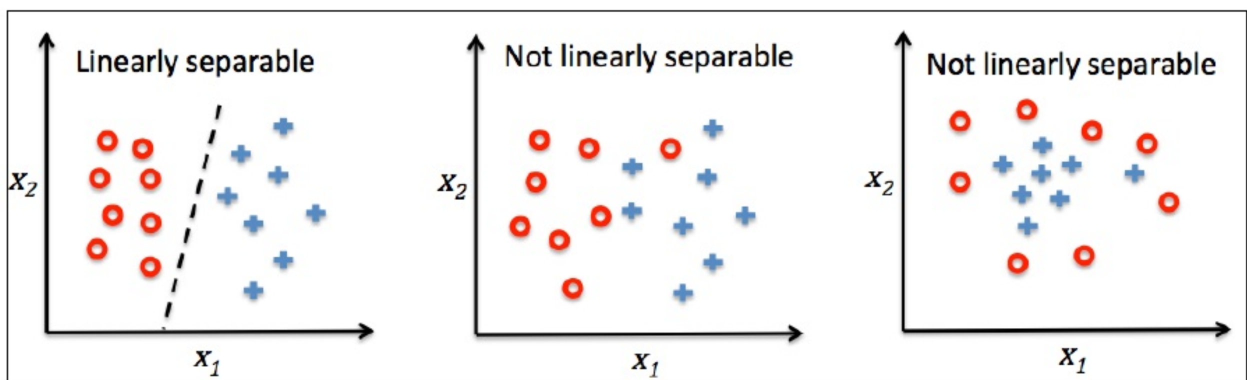
假设$x_j^{(i)} = 0.5$,我们将样本误分类为-1.此时，更新过程会对权重参数加1，下一次在对样本i计算输出值时，有更大的可能输出1：

$$\Delta w_j^{(i)} = \left(1^{(i)} - -1^{(i)}\right)0.5^{(i)} = (2)0.5^{(i)} = 1$$

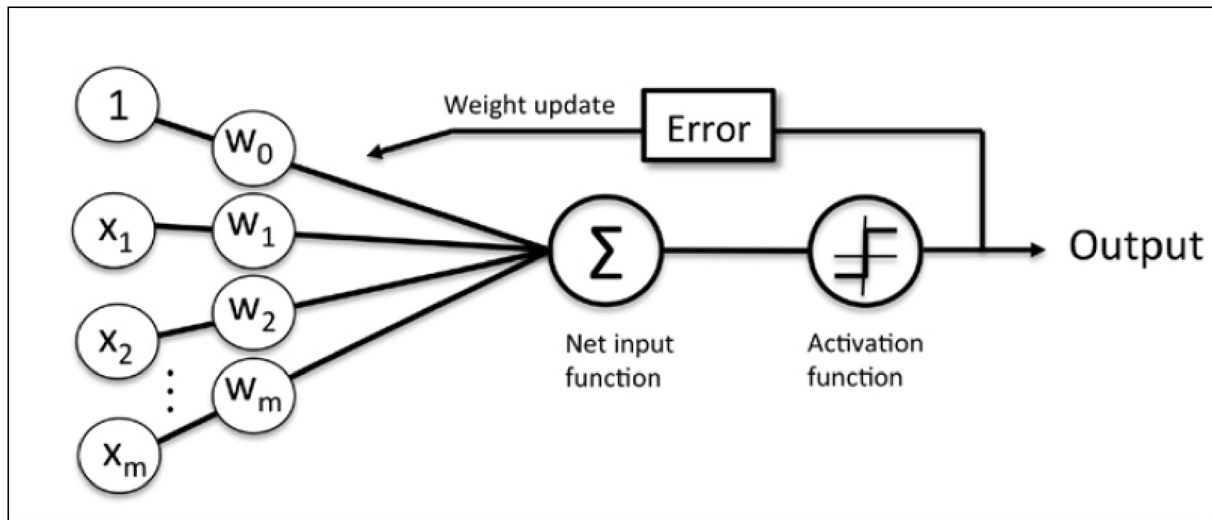
参数更新和样本$x_j^{(i)}$成正比。比如，如果我们有另一个样本$x_j^{(i)}$=2 被误分类为-1，在更新w_j时会朝着正确方法更新更多(相比较$x_j^{(i)}$=0.5的情况)：

$$\Delta w_j = \left(1^{(i)} - -1^{(i)}\right)2^{(i)} = (2)2^{(i)} = 4$$

感知机算法仅在两个类别确实线性可分并且学习率充分小的情况下才能保证收敛。如果两个类别不能被一个线性决策界分开，我们可以设置最大训练集迭代次数(epoch)或者可容忍的错误分类样本数 来停止算法的学习过程。



在进入下一节的代码实现之前，我们来总结一下感知机的要点：



感知机接收一个样本输入x，然后将其和权重w结合，计算网络输入z。z接着被传递给激活函数，产生一个二分类输出-1或1作为预测的样本类别。在整个学习阶段，输出用于计算预测错误率(y-□)和更新权重参数。

使用Python实现感知机算法

在前一节，我们学习了Rosenblatt的感知机如果工作；这一节我们用Python对其进行实现，并且应用于Iris数据集。关于代码的实现，我们使用面向对象的编程思想，定义一个感知机接口作为Python类，类中的方法主要有初始化方法，fit方法和predict方法。

```
import numpy as np

class Perceptron(object):
    """Perceptron classifier.

    Parameters
    ------------
    eta:float
        Learning rate (between 0.0 and 1.0)
    n_iter:int
        Passes over the training dataset.

    Attributes
    -------------
    w_: 1d-array
        Weights after fitting.
    errors_: list
        Numebr of misclassifications in every epoch.

    """

    def __init__(self, eta=0.01, n_iter=10):
        self.eta = eta
        self.n_iter = n_iter

    def fit(self, X, y):
        """Fit training data.

        Parameters
        ------------
        X: {array-like}, shape=[n_samples, n_features]
            Training vectors, where n_samples is the number of samples
            and n_featuers is the number of features.
        y: array-like, shape=[n_smaples]
            Target values.

        Returns
        ----------
        self: object
        """

        self.w_ = np.zeros(1 + X.shape[1]) # Add w_0
```

```
        self.errors_ = []

        for _ in range(self.n_iter):
            errors = 0
            for xi, target in zip(X, y):
                update = self.eta * (target - self.predict(xi))
                self.w_[1:] += update * xi
                self.w_[0] += update
                errors += int(update != 0.0)
            self.errors_.append(errors)
        return self

    def net_input(self, X):
        """Calculate net input"""
        return np.dot(X, self.w_[1:]) + self.w_[0]

    def predict(self, X):
        """Return class label after unit step"""
        return np.where(self.net_input(X) >= 0.0, 1, -1) #analoge ? : in C++
```

有了以上的代码实现，我们可以初始化一个新的Perceptron对象，并且对学习率eta和迭代次数n_iter赋值，fit方法先对权重参数初始化，然后对训练集中每一个样本循环，根据感知机算法对权重进行更新。类别通过predict方法进行预测。除此之外，self.errors_ 还记录了每一轮中误分类的样本数，有助于接下来我们分析感知机的训练过程。

基于Iris数据集训练感知机模型

我们使用Eirs数据集检验上面的感知机代码，由于我们实现的是一个二分类感知机算法，所以我们仅使用Iris中Setosa和Versicolor两种花的数据。为了简便，我们仅使用sepal length和petal length两维度的特征。记住，感知机模型不局限于二分类问题，可以用通过One-vs-All技巧扩展到多分类问题。

One-vs-All(OvA)有时也被称为**One-vs-Rest(OvR)**，是一种常用的将二分类分类器扩展为多分类分类器的技巧。通过OvA技巧，我们为每一个类别训练一个分类器，此时，对应类别为正类，其余所有类别为负类。对新样本数据进行类别预测时，我们使用训练好的所有类别模型对其预测，将具有最高置信度的类别作为最后的结果。对于感知机来说，最高置信度指的是网络输入z绝对值最大的那个类别。

回到刚才的Iris数据集，我们使用pandas读取数据，然后通过pandas中的tail方法输出最后五行数据，看一下Iris数据集格式:

```
In [1]: import pandas as pd

In [3]: df = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data', header=None)

In [4]: df.tail()
Out[4]:
```

	0	1	2	3	4
145	6.7	3.0	5.2	2.3	Iris-virginica
146	6.3	2.5	5.0	1.9	Iris-virginica
147	6.5	3.0	5.2	2.0	Iris-virginica
148	6.2	3.4	5.4	2.3	Iris-virginica
149	5.9	3.0	5.1	1.8	Iris-virginica

```
In [ ]:
```

接下来我们抽取出前100条样本，这正好是Setosa和Versicolor对应的样本，我们将Versicolor对应的数据作为类别1，Setosa对应的作为-1。对于特征，我们抽取出sepal length和petal length两维度特征，然后用散点图对数据进行可视化:

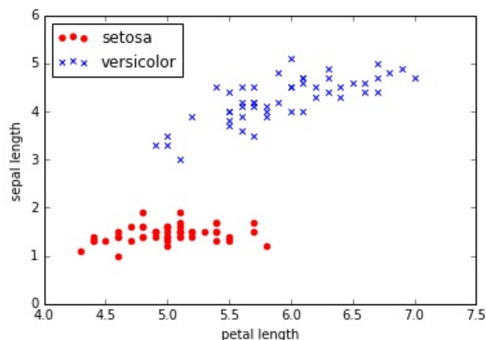
In [17]:
```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
```

In [18]:
```
y = df.iloc[0:100, 4].values
```

In [19]:
```
y = np.where(y == 'Iris-setosa', -1, 1)
```

In [20]:
```
X = df.iloc[0:100, [0, 2]].values
```

In [21]:
```
plt.scatter(X[:50, 0], X[:50, 1], color='red', marker='o', label='setosa')
plt.scatter(X[50:100, 0], X[50:100, 1], color='blue', marker='x', label='versicolor')
plt.xlabel('petal length')
plt.ylabel('sepal length')
plt.legend(loc='upper left')
plt.show()
```



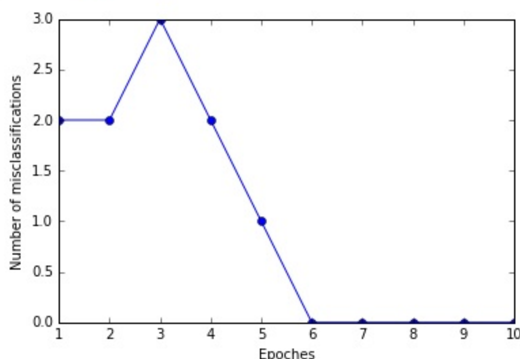
现在开始训练我们的感知机模型，为了更好地了解感知机训练过程，我们将每一轮的误分类数目可视化出来，检查算法是否收敛和找到分界线：

In [26]:
```
ppn = Perceptron(eta = 0.1, n_iter = 10)
```

In [27]:
```
ppn.fit(X, y)
```

Out[27]:
```
<__main__.Perceptron at 0xd52a2e8>
```

In [30]:
```
plt.plot(range(1, len(ppn.errors_) + 1), ppn.errors_, marker = 'o')
plt.xlabel('Epoches')
plt.ylabel('Number of misclassifications')
plt.show()
```



通过上图我们可以发现，第6次迭代时，感知机算法已经收敛了，对训练集的预测准群率是100%。接下来我们将分界线画出来：

In [31]:
```
from matplotlib.colors import ListedColormap
```

In [33]:
```
def plot_decision_region(X, y, classifier, resolution=0.02):
    #setup marker generator and color map
    markers = ('s', 'x', 'o', '^', 'v')
    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
    cmap = ListedColormap(colors[:len(np.unique(y))])

    #plot the decision surface
    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1

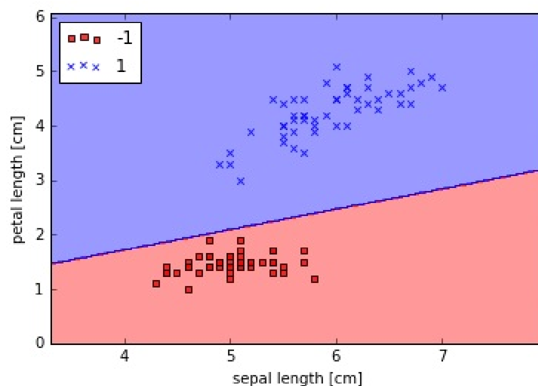
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                           np.arange(x2_min, x2_max, resolution))
    Z = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    Z = Z.reshape(xx1.shape)

    plt.contourf(xx1, xx2, Z, alpha=0.4, cmap=cmap)
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())

    #plot class samples

    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(x=X[y == cl, 0], y=X[y == cl, 1],
                    alpha=0.8, c=cmap(idx),
                    marker = markers[idx], label=cl)
```

In [35]:
```
plot_decision_region(X, y, classifier=ppn)
plt.xlabel('sepal length [cm]')
plt.ylabel('petal length [cm]')
plt.legend(loc='upper left')
plt.show()
```



虽然对于Iris数据集，感知机算法表现的很完美，但是"收敛"一直是感知机算法中的一大问题。Frank Rosenblatt从数学上证明了只要两个类别能够被一个现行超平面分开，则感知机算法一定能够收敛。然而，如果数据并非线性可分，感知计算法则会一直运行下去，除非我们人为设置最大迭代次数n_iter。

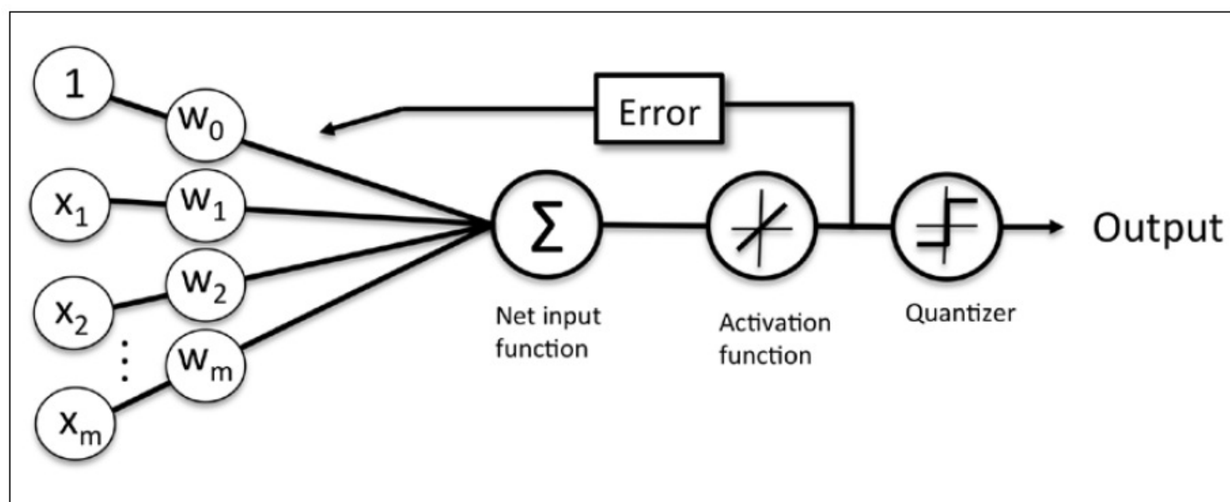
自适应线性神经元及收敛问题

本节我们学习另一种单层神经网络：自适应线性神经元(ADAptive LInear NEuron, 简称Adaline)。在Frank Rosenblatt提出感知计算法不久，Bernard Widrow和他的博士生Tedd Hoff提出了Adaline算法作为感知机的改进算法(B.Widrow et al. Adaptive "Adaline" neuron using chemical "memistors".)

相对于感知机，Adaline算法有趣的多，因为在学习Adaline的过程中涉及到机器学习中一个重要的概念：定义、最小化损失函数。学习Adaline为以后学习更复杂高端的算法(比如逻辑斯蒂回归、SVM等)起到抛砖引玉的作用。

Adaline和感知机的一个重要区别是Adaline算法中权重参数更新按照线性激活函数而不是单位阶跃函数。当然，Adaline中激活函数也简单的很，。

虽然Adaline中参数更新不是使用阶跃函数，但是在对测试集样本输出预测类别时还是使用阶跃函数，毕竟要输出离散值-1,1。



使用梯度下降算法最小化损失函数

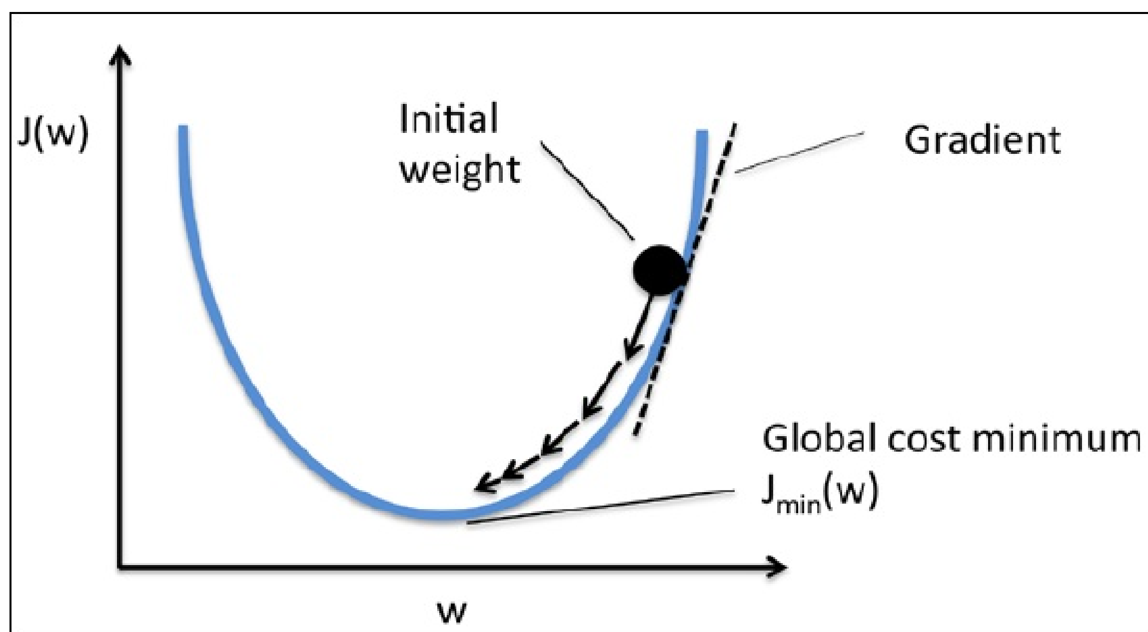
在监督机器学习算法中，一个重要的概念就是定义目标函数(objective function)，而目标函数就是机器学习算法的学习过程中要优化的目标，目标函数我们常称为损失函数(cost function)，在算法学习(即，参数更新)的过程中就是要最小化损失函数。

对于Adaline算法，我们定义损失函数为样本真实值和预测值之间的误差平方和(Sum of Squared Erros, SSE):

$$J(w)=\frac{1}{2}\sum_i\left(y^{(i)}-\phi\left(z^{(i)}\right)\right)^2.$$

上式中的系数完全是为了求导数方便而添加的，没有特殊的物理含义。相对于感知机中的单位阶跃函数，使用连续现行激活函数的一大优点是Adaline的损失函数是可导的。另一个很好的特性是Adaline的损失函数是凸函数，因为，我们可以使用简单而有效的优化算法：梯度下降(gradient descent)来找到使损失函数取值最小的权重参数。

如下图所示，我们可以把梯度下降算法看做"下山",直到遇到局部最小点或者全局最小点才会停止计算。在每一次迭代过程中，我们沿着梯度下降方向迈出一步，而步伐的大小由学习率和梯度大小共同决定。



使用梯度下降算法，实质就是运用损失函数的梯度来对参数进行更新：

$$w := w + \Delta w$$

此时，的值由负梯度乘以学习率确定：

$$\Delta w = -\eta \Delta J(w).$$

而要计算出损失函数的梯度，我们需要计算损失函数对每一个权重参数的偏导数w_j:

因此，.

注意所有权重参数还是同时更新的，所以Adaline算法的学习规则可以简写: .

虽然简写以后的学习规则和感知机一样，但不要忘了的不同。此外，还有一点很大的不同是在计算权重更新的过程中：Adaline需要用到所有训练集样本才能一次性更新所有的w，而感知机则是每次用一个训练集样本更新所有权重参数。所以梯度下降法常被称为批量梯度下降 ("batch" gradient descent)。

福利： 详细的损失函数对权重的偏导数计算过程为：

$$\frac{\partial J}{\partial w_j} = \frac{\partial}{\partial w_j} \frac{1}{2} \sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right) \right)^2$$

$$= \frac{1}{2} \frac{\partial}{\partial w_j} \sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right) \right)^2$$

$$= \frac{1}{2} \sum_i 2 \left(y^{(i)} - \phi\left(z^{(i)}\right) \right) \frac{\partial}{\partial w_j} \left(y^{(i)} - \phi\left(z^{(i)}\right) \right)$$

$$= \sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right) \right) \frac{\partial}{\partial w_j} \left(y^{(i)} - \sum_i \left(w_j^{(i)} x_j^{(i)} \right) \right)$$

$$= \sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right) \right) \left(-x_j^{(i)} \right)$$

$$= -\sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right) \right) x_j^{(i)}$$

Python实现自适应线性神经元

既然感知机和Adaline的学习规则非常相似，所以在实现Adaline的时候我们不需要完全重写，而是在感知机代码基础上进行修改得到Adaline，具体地，我们需要修改fit方法，实现梯度下降算法:

```
class AdalineGD(object):
    """ADAptive LInear NEuron classifier.

    Parameters
    ----------------
    eta: float
        Learning rate (between 0.0 and 1.0)
    n_iter: int
        Passes over the training dataset.

    Attributes:
    ------------------
    w_: 1d-array
        Weights after fitting.
    errors_: int
        Number of misclassification in every epoch.

    """

    def __init__(self, eta=0.01, n_iter=50):
        self.eta = eta
        self.n_iter = n_iter

    def fit(self, X, y):
        """Fit training data.

        Parameters
        ---------------
        X: {array-like}, shape=[n_samples, n_features]
            Training vectors,
        y: array-like, shape=[n_samples]
            Target values.

        Returns
        -----------
        self: object
        """
        self.w_ = np.zeros(1 + X.shape[1])
        self.cost_ = []

        for i in range(self.n_iter):
            output = self.net_input(X)
```

```
            errors = (y - output)
            self.w_[1:] += self.eta * X.T.dot(errors)
            self.w_[0] += self.eta * errors.sum()
            cost = (errors ** 2).sum() / 2.0
            self.cost_.append(cost)
        return self

    def net_input(self, X):
        """ Calculate net input"""
        return np.dot(X, self.w_[1:]) + self.w_[0]

    def activation(self, X):
        """ Compute linear activation"""
        return self.net_input(X)

    def predict(self, X):
        """ Return class label after unit step"""
        return np.where(self.activation(X) >= 0.0, 1, -1)
```

不像感知机那样每次用一个训练样本来更新权重参数，Adaline基于整个训练集的梯度来更新权重。

注意，**X.T.dot(errors)**是一个矩阵和向量的乘法，可见**numpy**做矩阵计算的方便性。

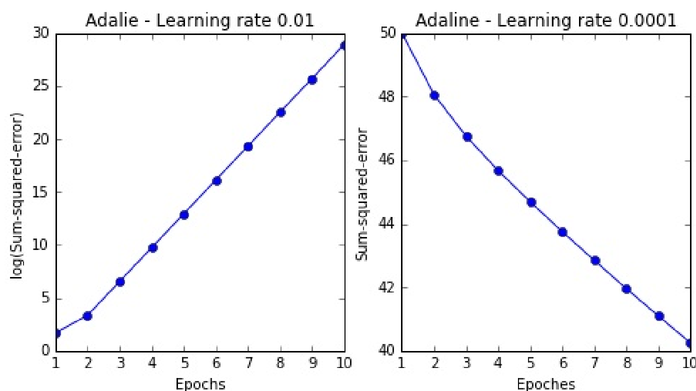
$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 7 \\ 8 \\ 9 \end{bmatrix} = \begin{bmatrix} 1\times7+2\times8+3\times9 \\ 4\times7+5\times8+6\times9 \end{bmatrix} = \begin{bmatrix} 50 \\ 122 \end{bmatrix}$$

在将Adaline应用到实际问题中时，通常需要先确定一个好的学习率这样才能保证算法真正收敛。我们来做一个试验，设置两个不同的值：$\eta = 0.01, \eta = 0.0001$。然后将每一轮的损失函数值画出来，窥探Adaline是如何学习的.

(学习率,迭代轮数n_iter也被称为超参数(hyperparameters),超参数对于模型至关重要，在第四章我们将学习一些技巧，能够自动找到能使模型达到最好效果的超参数。)

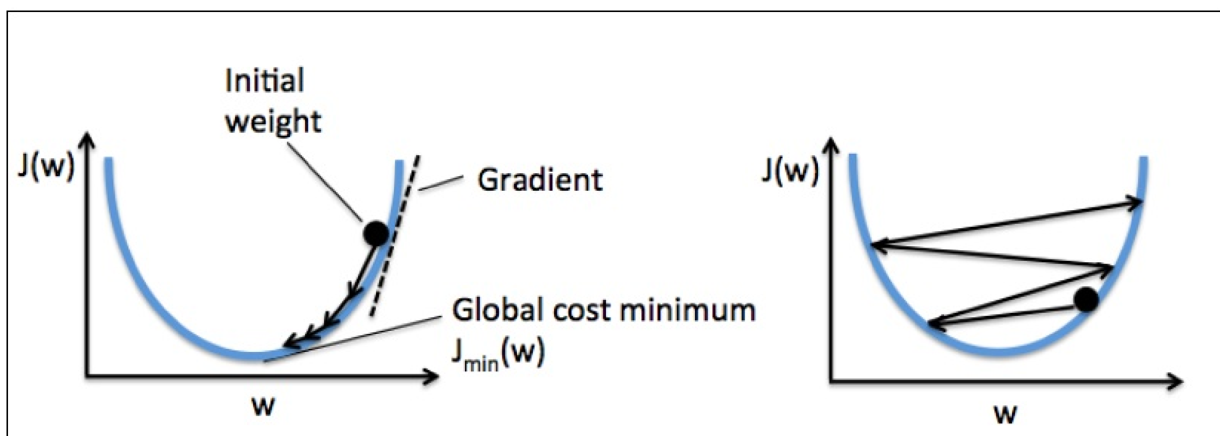
```
In [20]: fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(8,4))
         ada1 = AdalineGD(n_iter=10, eta=0.01).fit(X,y)
         ax[0].plot(range(1, len(ada1.cost_) + 1),
                    np.log10(ada1.cost_), marker='o')
         ax[0].set_xlabel('Epochs')
         ax[0].set_ylabel('log(Sum-squared-error)')
         ax[0].set_title('Adalie - Learning rate 0.01')

         ada2 = AdalineGD(n_iter=10, eta=0.0001).fit(X,y)
         ax[1].plot(range(1, len(ada2.cost_) + 1),
                    ada2.cost_, marker='o')
         ax[1].set_xlabel('Epoches')
         ax[1].set_ylabel('Sum-squared-error')
         ax[1].set_title('Adaline - Learning rate 0.0001')
         plt.show()
```



分析上面两幅图各自的问题，左图根本不是在最小化损失函数，反而在每一轮迭代过程中，损失函数值不断在增大！这说明取值过大的学习率不但对算法毫无益处反而危害大大滴。右图虽然能够在每一轮迭代过程中一直在减小损失函数的值，但是减小的幅度太小了，估计至少上百轮迭代才能收敛，而这个时间我们是耗不起的，所以学习率值过小就会导致算法收敛的时间巨长，使得算法根本不能应用于实际问题。

下面左图展示了权重再更新过程中如何得到损失函数$J(w)$最小值的。右图展示了学习率过大时权重更新，每次都跳过了最小损失函数对应的权重值。



许多机器学习算法都要求先对特征进行某种缩放操作，比如标准化(standardization)和归一化(normalization)。而缩放后的特征通常更有助于算法收敛，实际上，对特征缩放后在运用梯度下降算法往往会有更好的学习效果。

特征标准化的计算很简单，比如要对第j维度特征进行标准化，只需要计算所有训练集样本中第j维度的平均值u_j和标准差即可,然后套公式：

$$x'_j = \frac{x_j - \mu_j}{\sigma_j}$$

标准化后的特征 均值为0，标准差为1.

在Numpy中，调用mean和std方法很容易对特征进行标准化:

```
In  [36]: X_std = np.copy(X)
```

```
In  [37]: X_std[:, 0] = (X[:,0] - X[:,0].mean()) / X[:,0].std()
```

```
In  [38]: X_std[:, 1] = (X[:,1] - X[:,1].mean()) / X[:,1].std()
```

标准化后，我们用Adaline算法来训练模型，看看如何收敛的(学习率为0.01)：

```
In  [39]: ada = AdalineGD(n_iter=15, eta=0.01)
          ada.fit(X_std, y)

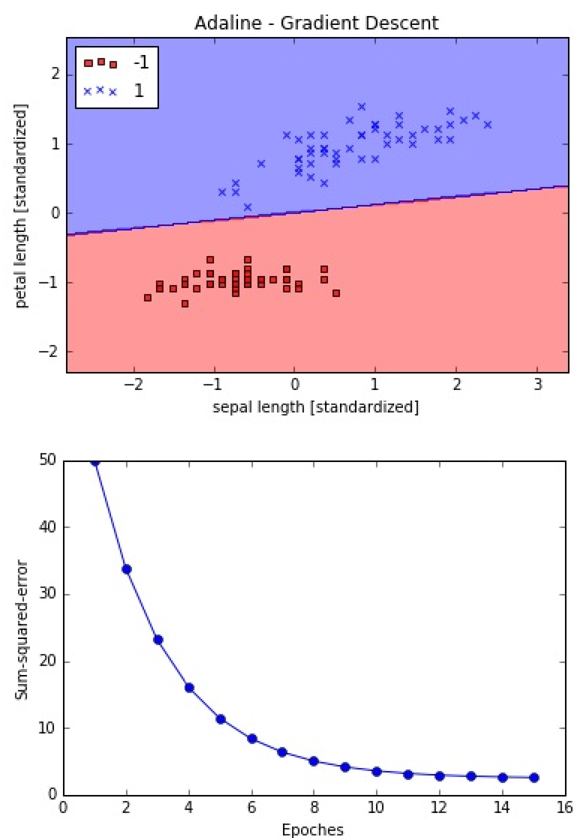
Out[39]: <__main__.AdalineGD at 0xe1690f0>
```

我们将决策界和算法学习情况可视化出来:

In [40]:
```
plot_decision_region(X_std, y, classifier=ada)

plt.title('Adaline - Gradient Descent')
plt.xlabel('sepal length [standardized]')
plt.ylabel('petal length [standardized]')
plt.legend(loc='upper left')
plt.show()

plt.plot(range(1, len(ada.cost_) + 1), ada.cost_, marker='o')
plt.xlabel('Epoches')
plt.ylabel('Sum-squared-error')
plt.show()
```



Wow 标准化后的数据再使用梯度下降Adaline算法竟然收敛了！ 注意看Sum-squared-error(即，$y - w^T x$)最后并没有等于0，即使所有样本都正确分类。

大规模机器学习和随机梯度下降

在上一节我们学习了如何使用梯度下降法最小化损失函数，由于梯度下降要用到所有的训练样本，因此也被称为批梯度下降(batch gradient descent)。现在想象一下我们有一个非常大的数据集，里面有几百万条样本，现在用梯度下降法来训练模型，可以想象计算量将是非常大，每一次求梯度都要用到所有的样本。能不能用少量的样本来求梯度呢？

随机梯度下降法(stochastic gradient descent)诞生啦！有时也被称为迭代(iteration)/在线(on-line)梯度下降。随机梯度下降法每次只用一个样本对权重进行更新(译者注：唔，感知机算法也如此,转了一圈，历史又回到了起点。)：

$$\eta\left(y^{(i)}-\phi\left(z^{(i)}\right)\right)\boldsymbol{x}^{(i)}$$

虽然随机梯度下降被当作是梯度下降的近似算法，但实际上她往往比梯度下降收敛更快，因为相同时间内她对权重更新的更频繁。由于单个样本得到的损失函数相对于用整个训练集得到的损失函数具有随机性，反而会有助于随机梯度下降算法避免陷入局部最小点。在实际应用随机梯度下降法时，为了得到准确结果，一定要以随机方式选择样本计算梯度，通常的做法在每一轮迭代后将训练集进行打乱重排(shuffle)。

Notes:在随机梯度下降法中，通常用不断减小的自适应学习率替代固定学习率 ,比如 ,其中c_1, c_2是常数。还要注意随机梯度下降并不能够保证使损失函数达到全局最小点，但结果会很接近全局最小。

随机梯度下降法的另一个优点是可以用于在线学习(online learning)。在线学习在解决不断累积的大规模数据时非常有用，比如，移动端的顾客数据。使用在线学习，系统可以实时更新并且如果存储空间快装不下数据了，可以将时间最久的数据删除。

Notes 除了梯度下降算法和随机梯度下降算法之外，还有一种常用的二者折中的算法：最小批学习(mini-batch learning)。很好理解，梯度下降每一次用全部训练集计算梯度更新权重，随机梯度法每一次用一个训练样本计算梯度更新权重，最小批学习每次用部分训练样本计算梯度更新权重，比如50。相对于梯度下降，最小批收敛速度也更快因为权重参数更新更加频繁。此外，最小批相对于随机梯度中，使用向量操作替代for循环(每一次跌倒都要遍历所有样本)，使得计算更快。

上一节我们已经实现了梯度下降求解Adaline，只需要做部分修改就能得到随机梯度下降法求解Adaline。第一个修改是fit方法内用每一个训练样本更新权重参数w,第二个修改是增加partial_fit方法，第三个修改是增加shuffle方法打乱训练集顺序。

```
from numpy.random import seed
```

```
class AdalineSGD(object):
    """ADAptive LInear NEuron classifier.

    Parameters
    ---------------
    eta: float
        Learning rate (between 0.0 and 1.0)
    n_iter: int
        Passes over the training dataset.

    Attributes
    ---------------
    w_: 1d-array
        Weights after fitting.
    errors_: list
        Number of misclassification in every epoch.
    shuffle: bool (default: True)
        Shuffles training data every epoch
        if True to prevent cycles.
    random_state: int (default: None)
        Set random state for shuffling
        and initializing the weights.

    """
    def __init__(self, eta=0.01, n_iter=10, shuffle=True, random_state=None):
        self.eta = eta
        self.n_iter = n_iter
        self.w_initialized = False
        self.shuffle = shuffle
        if random_state:
            seed(random_state)

    def fit(self, X, y):
        """Fit training data.

        Parameters
        -----------
        X: {array-like}, shape=[n_samples, n_features]
        y: array-like, shape=[n_samples]

        Returns
        --------------
        self: object
        """
        self._initialize_weights(X.shape[1])
        self.cost_ = []
        for i in range(self.n_iter):
            if self.shuffle:
                X, y = self._shuffle(X, y)
            cost = []
            for xi, target in zip(X, y):
                cost.append(self._update_weights(xi, target))
```

```
            avg_cost = sum(cost)/len(y)
            self.cost_.append(avg_cost)
        return self

    def partial_fit(self, X, y):
        """Fit training data without reinitializing the weights."""
        if not self.w_initialized:
            self._initialize_weights(X.shape[1])
        if y.ravel().shape[0] > 1:
            for xi, target in zip(X, y):
                self._update_weights(xi, target)
        else:
            self._update_weights(X, y)
        return self

    def _shuffle(self, X, y):
        """Shuffle training data"""
        r = np.random.permutation(len(y))
        return X[r], y[r]

    def _initialize_weights(self, m):
        """Initialize weights to zeros"""
        self.w_ = np.zeros(1 + m)
        self.w_initialized = True

    def _update_weights(self, xi, target):
        """Apply Adaline learning rule to update the weights"""
        output = self.net_input(xi)
        error = (target - output)
        self.w_[1:] += self.eta * xi.dot(error)
        self.w_[0] += self.eta * error
        cost = 0.5 * error ** 2
        return cost

    def net_input(self, X):
        """Calculate net input"""
        return np.dot(X, self.w_[1:]) + self.w_[0]

    def activation(self, X):
        """Compute linear activation"""
        return self.net_input(X)

    def predict(self, X):
        """Return class label after unit step"""
        return np.where(self.activation(X) >= 0.0, 1, -1)
```

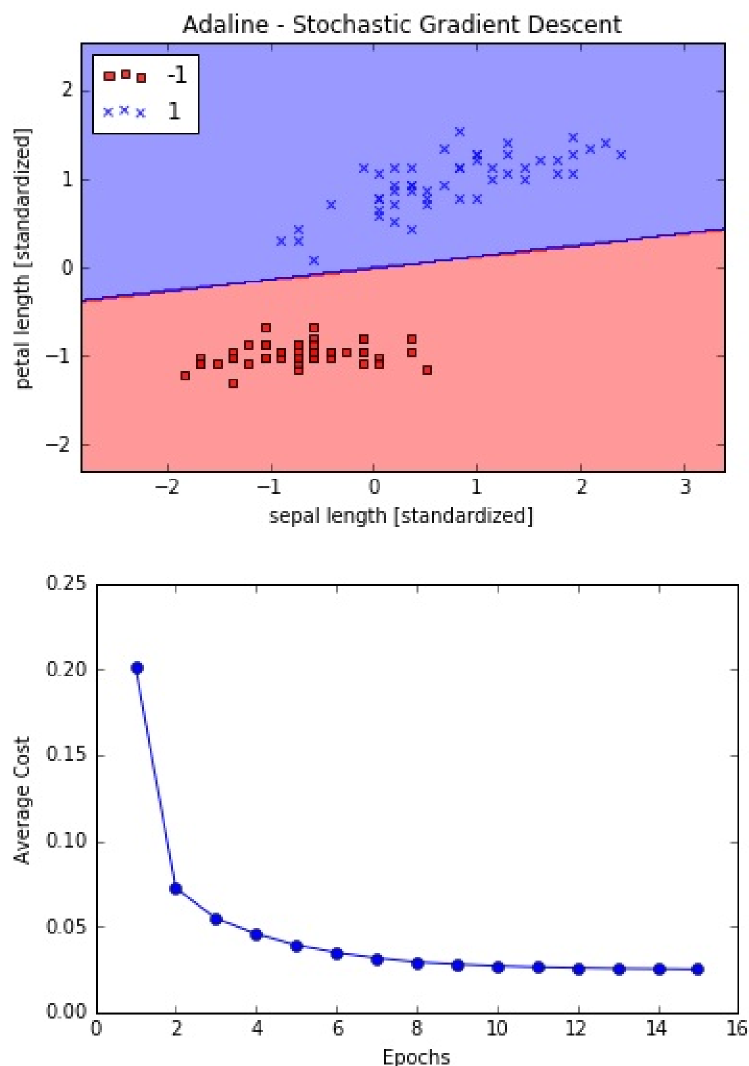
_shuffle方法的工作方式：调用numpy.random中的permutation函数得到0-100的一个随机序列，然后这个徐列作为特征矩阵和类别向量的下标，就起到了shuffle的功能。

我们使用fit方法训练AdalineSGD模型，使用plot_decision_regions对训练结果画图:

```
In [21]: ada = AdalineSGD(n_iter=15, eta=0.01, random_state=1)
         ada.fit(X_std, y)

Out[21]: <__main__.AdalineSGD at 0xc3bc780>
```

```
In [23]: plot_decision_region(X_std, y, classifier=ada)
         plt.title('Adaline - Stochastic Gradient Descent')
         plt.xlabel('sepal length [standardized]')
         plt.ylabel('petal length [standardized]')
         plt.legend(loc = 'upper left')
         plt.show()
         plt.plot(range(1, len(ada.cost_) + 1), ada.cost_, marker='o')
         plt.xlabel('Epochs')
         plt.ylabel('Average Cost')
         plt.show()
```



我们可以发现，平均损失(average cost)下降的非常快，在第15次迭代后决策界和使用梯度下降的Adaline决策界非常相似。如果我们要在在线环境下更新模型参数，通过调用partial_fit方法即可，此时参数是一个训练样本，比如ada.partial_fit(X_std[0, :], y[0])。

Summary

第三章 使用Scikit-learn进行分类器之旅

本章我们将要学习学术界和工业界常用的几种机器学习算法。在学习算法之间差异的同时，我们也要了解每个算法的优缺点。我们不会像上一章亲自实现各个算法，而是直接调用易用的scikit-learn库。

本章涉及到的知识点包括：

- 介绍常用分类算法的概念
- 学习使用scikit-learn
- 如何选择机器学习算法

如何选择合适的分类器算法

对于一个具体的分类问题，如何选择合适的分类器算法绝非纸上谈兵就能确定的：每一个算法都有其独特的数据偏好和假设。再一次强调"No Free Lunch"定理：在所有场景下都最厉害的分类器是不存在滴。实际上，常用的做法就是多选择几个分类器试试，然后挑选效果最好的那一个。对于同一个问题，样本数的不同、特征数目的多少、数据集中的噪音和数据是否线性可分 都会影响到分类器的效果。

最终，分类器的性能、计算能力和预测能力，都极大依赖训练集。我们概况一下训练一个机器学习算法通常的5大步骤：

- 特征选择
- 选择性能评价指标
- 选择分类器和优化算法
- 评估模型的性能
- 模型调参

本书的内容贯穿上面的5大步骤，不要心急，本章重点关注常用算法的重要概念，至于特征选择、预处理、评价指标和如何调参将在后面章节一一介绍。

scikit-learn之旅

在第二章，你学习了两个分类相关的算法：感知机和Adaline，并且都用Python进行了实现。现在我们学习scikit-learn的API，这个库不但用户界面友好并且对常用算法的实现进行了高度优化。此外，它还包含数据预处理和调参和模型评估的很多方法，是Python进行数据挖掘的必备工具。

通过scikit-learn训练感知机模型

我们先看一下如何使用sklearn训练一个感知机模型，数据集还是用我们熟悉的Iris数据集。由于sklearn已经内置了Iris数据集，所以本章所有的分类算法我们通通使用Iris数据集，还是和第二章一样，为了可视化方便，我们只使用其中两维度特征，样本数则使用三个类别全部的150个样本。

```
In [1]: %matplotlib inline
        import numpy as np
        from sklearn import datasets
```

```
In [2]: iris = datasets.load_iris()
        X = iris.data[:, [2, 3]]
        y = iris.target
```

```
In [3]: np.unique(y)
Out[3]: array([0, 1, 2])
```

为了评估训练好的模型对新数据的预测能力，我们先把Iris训练集分割为两部分：训练集和测试集。在第5章我们还会讨论模型评估的更多细节。

```
In [4]: from sklearn.cross_validation import train_test_split
        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=0)
```

通过调用train_tset_split方法我们将数据集随机分为两部分，测试集占30%(45个样本)，训练集占70%(105个样本)。

许多机器学习和优化算法都要求对特征进行缩放操作，回忆第二章中梯度下降的例子，当时我们自己实现了标准化代码，现在我们可以直接调用sklearn中的StandardScaler来对特征进行标准化：

```
In [5]: from sklearn.preprocessing import StandardScaler
        sc = StandardScaler()
        sc.fit(X_train)
        X_train_std = sc.transform(X_train)
        X_test_std = sc.transform(X_test)
```

上面的代码中我们先从preprocessing模块中读取StandardScaler类，然后得到一个初始化的StandardScaler新对象sc，使用fit方法，StandardScaler对训练集中每一维度特征计算出u(样本平均值)和 (标准差)，然后调用transform方法对数据集进行标准化。注意我们用相同的标准化参数对待训练集和测试集。

对数据标准化以后，我们可以训练一个感知机模型。sklearn中大多数算法都支持多类别分类，默认使用One-vs.-Rest方式实现。所以我们可以直接训练三分类的感知机模型:

```
In [7]: from sklearn.linear_model import Perceptron

In [8]: ppn = Perceptron(n_iter=40, eta0=0.1, random_state=0)

In [9]: ppn.fit(X_train_std, y_train)

Out[9]: Perceptron(alpha=0.0001, class_weight=None, eta0=0.1, fit_intercept=True,
              n_iter=40, n_jobs=1, penalty=None, random_state=0, shuffle=True,
              verbose=0, warm_start=False)
```

sklearn中训练模型的接口和我们在第二章实现的一样耶，从linear_model模型读取Perceptron类，然后初始化得到ppn，接着使用fit方法训练一个模型。这里的eta0就是第二章中的学习率eta，n_iter同样表示对训练集迭代的次数。我们设置random_state参数使得shuffle结果可再现。

训练好感知机模型后，我们可以使用predict方法进行预测了：

```
In [10]: y_pred = ppn.predict(X_test_std)

In [11]: print('Misclassified samples: %d' % (y_test != y_pred).sum())

         Misclassified samples: 4
```

对于测试集中45个样本，有4个样本被错分类了，因此，错分类率是0.089。除了使用错分类率，我们也可以使用分类准确率(accuracy)评价模型，accuracy=1-miscassification error = 1-0.089=0.911。

Sklearn中包含了许多评价指标，这些指标都位于metrics模块，比如，我们可以计算分类准确率:

```
In [20]: from sklearn.metrics import accuracy_score

In [21]: print('Accuracy: %.2f' % accuracy_score(y_test, y_pred))

         Accuracy: 0.91
```

Notes 本章我们评估模型性能的好坏仅仅依赖于其在测试集的表现。在第5章，你将会学习许多其他的技巧来评估模型，包括可视化分析来检测和预防过拟合(overfitting)。过拟合意味着模型对训练集中的模式捕捉的很好，但是其泛化能力却很差。

最后，我们使用第二章的plot_decision_regions画出分界区域。不过在使用之间，我们进行一点小修改，我们用圆圈表示测试集样本：

```
from matplotlib.colors import ListedColormap
import matplotlib.pyplot as plt

def plot_decision_regions(X, y, classifier, test_idx=None, resolution=0.02):
    # setup marker generator and color map
    markers = ('s', 'x', 'o', '^', 'v')
    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
    cmap = ListedColormap(colors[:len(np.unique(y))])

    # plot the decision surface
    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                           np.arange(x2_min, x2_max, resolution))
    Z = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    Z = Z.reshape(xx1.shape)
    plt.contourf(xx1, xx2, Z, slpha=0.4, cmap=cmap)
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())

    # plot all samples
    X_test, y_test = X[test_idx, :], y[test_idx]
    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(x=X[y == cl, 0], y=X[y == cl, 1],
                    alpha=0.8, c=cmap(idx),
                    marker=markers[idx], label=cl)

    # highlight test samples
    if test_idx:
        X_test, y_test = X[test_idx, :], y[test_idx]
        plt.scatter(X_test[:, 0], X_test[:, 1], c='',
                    alpha=1.0, linewidth=1, marker='o',
                    s=55, label='test set')
```

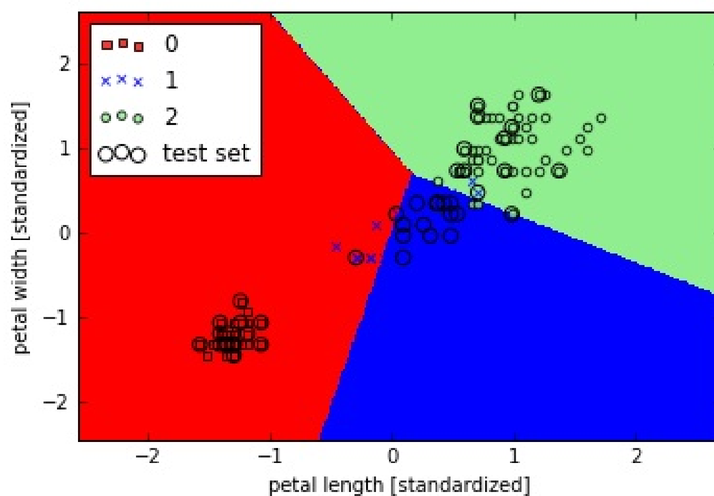
经过上面的修改，现在可以区分训练集和测试集，运行如下代码：

In [22]:
```
X_combined_std = np.vstack((X_train_std, X_test_std))
```

In [23]:
```
y_combined = np.hstack((y_train, y_test))
```

In [26]:
```
plot_decision_regions(X=X_combined_std,
                      y=y_combined,
                      classifier=ppn,
                      test_idx=range(105, 150))
plt.xlabel('petal length [standardized]')
plt.ylabel('petal width [standardized]')
plt.legend(loc='upper left')
plt.show()
```

对结果可视化后，我们可以发现三个类别的花不能被线性决策界完美分类：



在第二章我们就说过，感知机对于不能够线性可分的数据，算法永远不会收敛，这也是为什么我们不推荐大家实际使用感知机的原因。在接下来的章节，我们将学习到其他线性分类器，即使数据不能完美线性分类，也能收敛到最小的损失值。

逻辑斯蒂回归对类别概率建模

感知机算法为我们学习机器学习分类问题曾经立下汗马功劳，但由于其致命缺点：如果数据不能完全线性分割，则算法永远不会收敛。我们实际上很少真正使用感知机模型。

接下来我们学习另一个非常有效的线性二分类模型：逻辑斯蒂回归(logistic regression)。注意，尽管模型名字中有“回归”的字眼，但她确是百分百的分类模型。

逻辑斯蒂回归和条件概率

逻辑斯蒂回归(以下简称逻辑回归)是一个分类模型，它易于实现，并且对于线性可分的类别数据性能良好。她是工业界最常用的分类模型之一。和感知机和Adaline相似，本章的逻辑回归模型也是用于二分类的线性模型，当然可以使用OvR技巧扩展为多分类模型。

逻辑回归作为一个概率模型，为了解释其背后的原理，我们先介绍一个概念：几率(odds ratio)=, 其中p表示样本为正例的概率。这里如何划分正例、负例要根据我们想要预测什么，比如，我们要预测一个病人有某种疾病的概率，则病人有此疾病为正例。数学上，正例表示类别y=1。有了几率的概念，我们可以定义对数几率函数(logit function,这里log odds简称logit)：

对数几率函数的自变量p取值范围[0,1],因变量值域为实数域，将上式中的p视为类后验概率估计p(y=1|x),然后定义如下线性关系：

$$logit\left(p\left(y=1\mid \boldsymbol{x}\right)\right)=w_0x_0+w_1x_1+w_mx_m=\sum_{i=0}^{n}w_mx_m=\boldsymbol{w}^Tx$$

实际上我们关心的是某个样本属于类别的概率，恰恰是对数几率函数的反函数，也被称为逻辑斯底函数(logistic function)，有时简写为sigmoid函数，函数图像是S型：

$$\phi(z)=\frac{1}{1+e^{-z}}$$

其中z是网络输入，即权重参数和特征的线性组合 $z=w^Tx=w_0+w_1x_1+\ldots+w_mx_m$。

sigmoid(S曲线)函数很重要，我们不妨画图看一看：

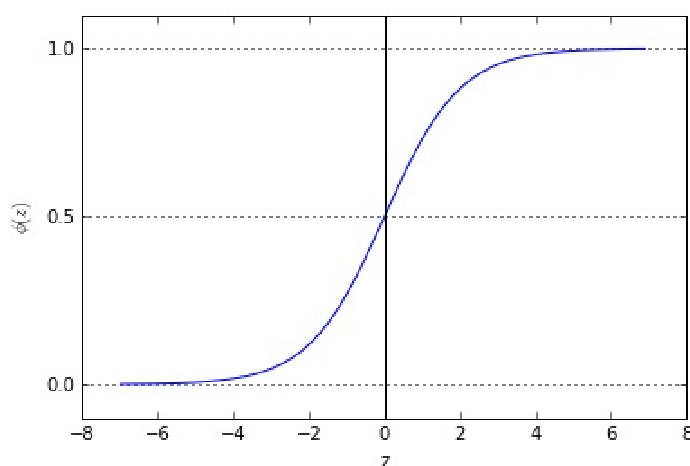
In [32]:
```
import matplotlib.pyplot as plt
import numpy as np
```

In [33]:
```
def sigmoid(z):
    return 1.0 / (1.0 + np.exp(-z))

z = np.arange(-7, 7, 0.1)
```

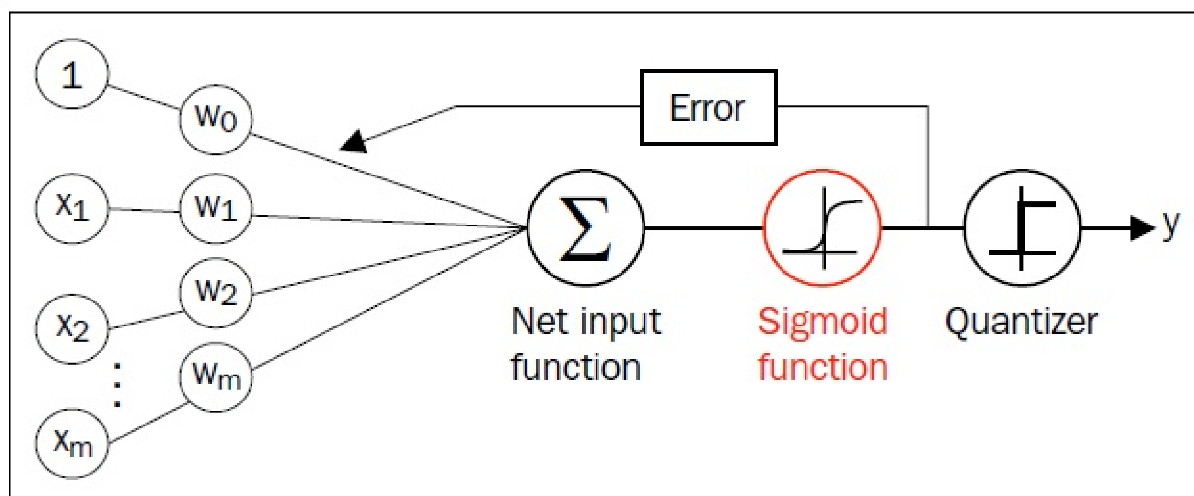
In [34]:
```
phi_z = sigmoid(z)
```

In [36]:
```
plt.plot(z, phi_z)
plt.axvline(0.0, color='k')
plt.axhspan(0.0, 1.0, facecolor='1.0', alpha=1.0, ls='dotted')
plt.axhline(y=0.5, ls='dotted', color='k')
plt.yticks([0.0, 0.5, 1.0])
plt.ylim(-0.1, 1.1)
plt.xlabel('z')
plt.ylabel('$\phi (z)$')
plt.show()
```



我们可以看到随着z趋向正无穷$z \to \infty$，无限接近于1；z趋向负无穷，无限接近0.因此，对于sigmoid函数，其自变量取值范围为实数域，因变量值域为[0,1],并且sigmoid(0)=0.5。

为了直观上对逻辑回归有更好的理解，我们可以和Adaline模型联系起来，二者的唯一区别是：**Adaline**模型，激活函数，在逻辑回归中，激活函数变成了**sigmoid**函数。



由于sigmoid函数的输出是在[0,1]，所以可以赋予物理含义：样本属于正例的概率，。举例来说，如果$\phi(z) = 0.8$，意味着此样本是Iris-Versicolor花的概率是0.8，是Iris-Setosa花的概率是$P(y=0|x;w) = 1 - P(y=1|x;w) = 0.2$。

有了样本的预测概率，再得到样本的类别值就很简单了,和Adaline一样，使用单位阶跃函数：

$$\hat{y} = \begin{cases} 1 & if\, \phi(z) \geq 0.5 \\ 0 & otherwise \end{cases}$$

上式等价于：

$$\hat{y} = \begin{cases} 1 & if\ z \geq 0.0 \\ 0 & otherwise \end{cases}$$

逻辑回归之所以应用广泛，一大优点就是它不但能预测类别，还能输出具体的概率值，概率值很很多场景往往比单纯的类别值重要的多。比如在天气预测中下雨的可能性，病人患病的可能性等等。

学习逻辑斯底损失函数中的权重参数

对逻辑回归模型有了基本认识后，我们回到机器学习的核心问题，怎样学习参数。还记得上一章Adaline中我们定义的差平方损失函数：

$$J(\boldsymbol{w})=\sum_i \frac{1}{2}\left(\phi\left(z^{(i)}\right)-y^{(i)}\right)^2$$

我们求解损失函数最小时的权重参数，同样，对于逻辑回归，我们也需要定义损失函数，在这之前，先定义似然(likelihood)L的概念，假设训练集中样本独立，似然定义：

$$L(\boldsymbol{w})=P(\boldsymbol{y}\mid\boldsymbol{x};\boldsymbol{w})=\prod_{i=1}^{n}P\left(y^{(i)}\mid x^{(i)};\boldsymbol{w}\right)=\left(\phi\left(z^{(i)}\right)\right)^{y^{(i)}}\left(1-\phi\left(z^{(i)}\right)\right)^{1-y^{(i)}}$$

与损失函数费尽全力找最小值相反，对于似然函数，我们要找的是最大值。实际上，对于似然的log值，是很容易找到最大值的，也就是最大化log-likelihood函数：

$$l(\boldsymbol{w})=\log L(\boldsymbol{w})=\sum_{i=1}^{n}\log\left(\phi\left(z^{(i)}\right)\right)+\left(1-y^{(i)}\right)\log\left(1-\phi\left(z^{(i)}\right)\right)$$

接下来，我们可以运用梯度下降等优化算法来求解最大化log-likelihood时的参数。最大化和最小化本质上没有区别，所以我们还是将log-likelihood写成求最小值的损失函数形式：

$$J(\boldsymbol{w})=\sum_{i=1}^{n}-\log\left(\phi\left(z^{(i)}\right)\right)-\left(1-y^{(i)}\right)\log\left(1-\phi\left(z^{(i)}\right)\right)$$

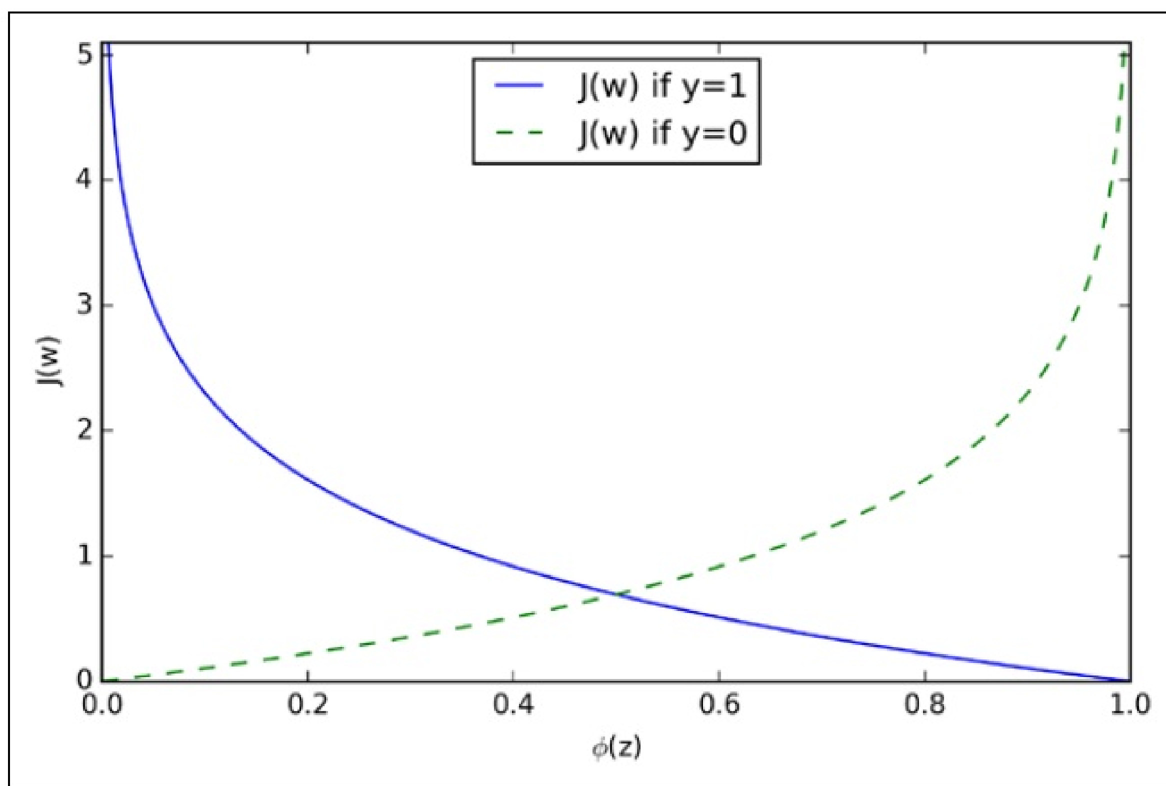
为了更好地理解此损失函数，假设现在训练集只有一个样本：

$$J(\phi(z),y;\boldsymbol{w})=-y\log(\phi(z))-(1-y)\log(1-\phi(z))$$

上式等号右边的算式，如果y=0则第一项为0；如果y=1则第二项为0。

$$J(\phi(z),y;\boldsymbol{w})=\begin{cases}-\log(\phi(z)) & if\ y=1\\ -\log(1-\phi(z)) & if\ y=0\end{cases}$$

下图展示了一个训练样本时，不同 时对应的$J(w)$:



对于蓝线，如果逻辑回归预测结果正确，类别为1，则损失为0；对于绿线，如果逻辑回归预测正确，类别为0，则损失为0。 如果预测错误，则损失趋向正无穷。

调用scikit-learn训练逻辑回归模型

如果我们自己实现逻辑回归，只需要将第二章中的Adaline中的损失函数替换掉即可，新的损失函数：

$$J(\boldsymbol{w}) = -\sum_i y^{(i)} \log\left(\phi\left(z^{(i)}\right)\right) + \left(1 - y^{(i)}\right) \log\left(1 - \phi\left(z^{(i)}\right)\right)$$

不过考虑到sklearn中提供了高度优化过的逻辑回归实现，同时也支持多类别分类，我们就不自己实现了，而是直接调用sklearn.linear_model.LogisticRegression，使用标准化后的Iris数据集训练模型：

```
In [ ]:
```

```
In [ ]:
```

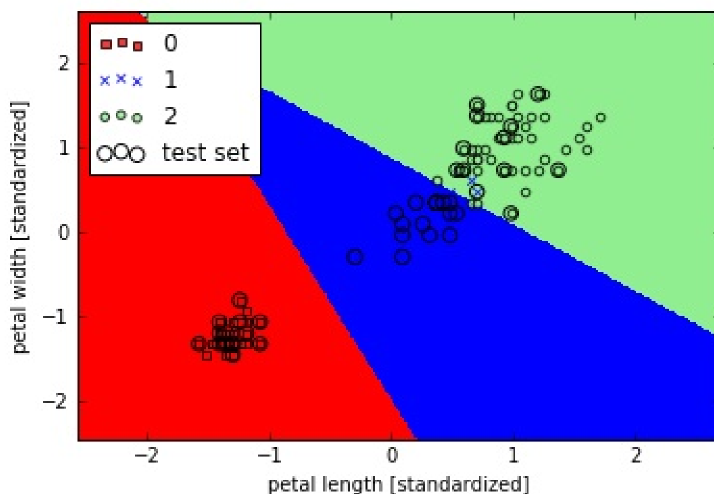
```
In [7]: from sklearn.linear_model import LogisticRegression
```

```
In [8]: lr = LogisticRegression(C=1000.0, random_state=0)
        lr.fit(X_train_std, y_train)

Out[8]: LogisticRegression(C=1000.0, class_weight=None, dual=False,
                  fit_intercept=True, intercept_scaling=1, max_iter=100,
                  multi_class='ovr', n_jobs=1, penalty='l2', random_state=0,
                  solver='liblinear', tol=0.0001, verbose=0, warm_start=False)
```

```
In [12]: plot_decision_regions(X_combined_std, y_combined, classifier=lr, test_idx=range(105, 150))
         plt.xlabel('petal length [standardized]')
         plt.ylabel('petal width [standardized]')
         plt.legend(loc='upper left')
         plt.show()
```

训练模型后，我们画出了决策界，



细心的你一定会问在初始化LogisticRegression时那个参数C是什么意思呀？在下一节讲正则化的时候我们再好好讨论讨论。

有了逻辑回归模型，我们就可以预测了，如果你想知道输出概率，调用predict_proba方法即可，

```
In [13]: lr.predict_proba(X_test_std[0, :])

c:\python27\lib\site-packages\sklearn\utils\validation.py:386: Dep
aise ValueError in 0.19. Reshape your data either using X.reshape(-
s a single sample.
  DeprecationWarning)

Out[13]: array([[  2.05743774e-11,   6.31620264e-02,   9.36837974e-01]])
```

上面的array表示lr认为测试样本属于Iris-Virginica类别的概率为93.683%，属于Iris-Versicolor的概率为6.316%。

不论Adaline还是逻辑回归，使用梯度下降算法更新权重参数时，用到的算式都一样，。

我们好好推导一下这个计算过程，首先计算log-likelihood函数对权重参数的偏导数：

$$\frac{\partial}{\partial w_j} l(\boldsymbol{w}) = \left(y \frac{1}{\phi(z)} - (1-y) \frac{1}{1-\phi(z)} \right) \frac{\partial}{\partial w_j} \phi(z)$$

然后，计算sigmoid函数的导数：

$$\frac{\partial}{\partial z} \frac{1}{1+e^{-z}} = \frac{1}{\left(1+e^{-z}\right)^2} e^{-z} = \frac{1}{1+e^{-z}} \left(1 - \frac{1}{1+e^{-z}}\right)$$
$$= \phi(z)\left(1-\phi(z)\right)$$

将上式代入到损失函数偏导数：

$$\left(y \frac{1}{\phi(z)} - (1-y) \frac{1}{1-\phi(z)} \right) \frac{\partial}{\partial w_j} \phi(z)$$
$$= \left(y \frac{1}{\phi(z)} - (1-y) \frac{1}{1-\phi(z)} \right) \phi(z)\left(1-\phi(z)\right) \frac{\partial}{\partial w_j} z$$
$$= \left(y\left(1-\phi(z)\right) - (1-y)\phi(z) \right) x_j$$
$$= \left(y - \phi(z) \right) x_j$$

我们的目标是求解使log-likelihood最大值时的参数w，因此，对于参数更新，我们按照：

$$w_j := w_j + \eta \sum_{i=1}^{n} \left(y^{(i)} - \phi\left(z^{(i)}\right) \right) x^{(i)}$$

由于每次更新权重时，对于所有参数同时更新，所以可以写成向量形式：

$$\boldsymbol{w} := \boldsymbol{w} + \Delta \boldsymbol{w}$$

我们定义

$$\Delta \boldsymbol{w} = \eta \nabla l(\boldsymbol{w})$$

由于最大化log-likelihood等价于最小化损失函数J,我们可以将梯度下降算法的权重更新写作：

$$\Delta w_j = -\eta \frac{\partial J}{\partial w_j} = \eta \sum_{i=1}^{n} \left(y^{(i)} - \phi\left(z^{(i)}\right) x^{(i)} \right)$$

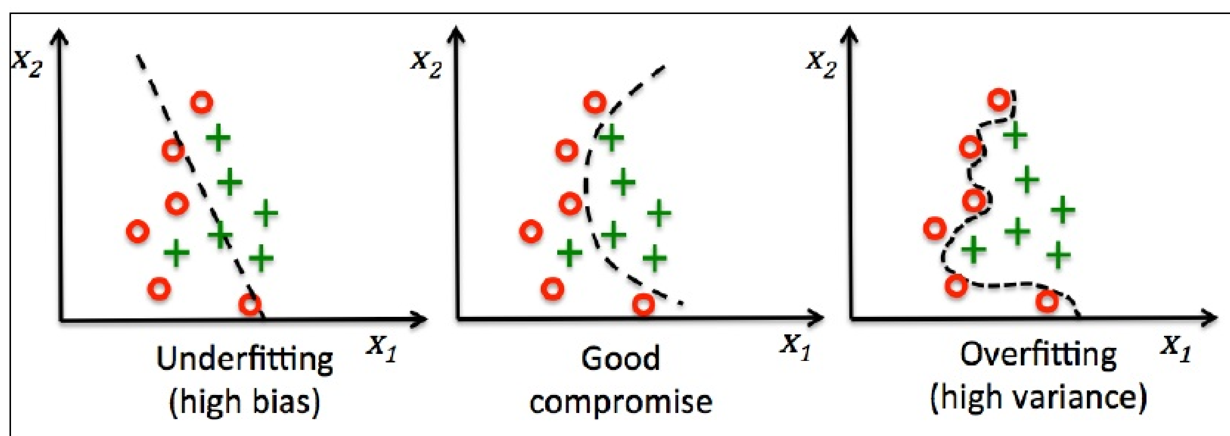
$$\boldsymbol{w} := \boldsymbol{w} + \Delta \boldsymbol{w},\ \Delta \boldsymbol{w} = -\eta \nabla J(\boldsymbol{w})$$

这和Adaline中的权重更新算式完全相同。

使用正则化解决过拟# 使用正则化解决过拟 # 使用正则化解决过拟# 使用正则化解决过拟 合

过拟合(overfitting)是机器学习中很常见的问题，指的是一个模型在训练集上表现很好但是泛化能力巨差(在测试集上的表现糟糕)。如果一个模型饱受过拟合困扰，我们也说此模型方差过高，造成这个结果的原因可能是模型含有太多参数导致模型过于复杂。同样，模型也可能遇到欠拟合(underfitting)问题，我们也说此模型偏差过高，原因是模型过于简单不能学习到训练集中数据存在的模式，同样对于测试集表现很差。

虽然到目前为止我们仅学习了用于分类任务的几种线性模型，过拟合和欠拟合问题可以用一个非线性决策界很好的演示：



怎样找到bias-variance之间的平衡，常用的方法是正则化(regularization)。正则化是解决特征共线性、过滤数据中噪音和防止过拟合的有用方法。正则化背后的原理是引入额外的信息(偏差)来惩罚过大的权重参数。最常见的形式就是所谓的L2正则(L2 regularization,有时也被称为权重衰减，L2收缩)：

$$\frac{\lambda}{2}\|\boldsymbol{w}\|^2 = \frac{\lambda}{2}\sum_{j=1}^{m} w_j^2$$

此处的λ就是正则化系数。

Note 正则化是为什么特征缩放如此重要的另一个原因。为了正则化起到作用，我们需要保证所有的特征都在可比较范围(comparable scales)。

如何应用正则化呢？我们只需要在现有损失函数基础上添加正则项即可，比如对于逻辑回归模型，带有L2正则项的的损失函数：

$$J(\boldsymbol{w})=\left[\sum_{i=1}^{n}\left(-\log\left(\phi\left(z^{(i)}\right)\right)+\left(1-y^{(i)}\right)\right)\left(-\log\left(1-\phi\left(z^{(i)}\right)\right)\right)\right]+\frac{\lambda}{2}\|\boldsymbol{w}\|^{2}$$

通过正则系数,我们可以控制在训练过程中使得参数w比较小。值越大，正则化威力越强大。

现在我们可以解释LogisticRegression中的参数C:

$$C=\frac{1}{\lambda}$$

所以，我们可以将逻辑回归 正则化的损失函数重写为

$$J(\boldsymbol{w})=C\left[\sum_{i=1}^{n}\left(-\log\left(\phi\left(z^{(i)}\right)\right)+\left(1-y^{(i)}\right)\right)\left(-\log\left(1-\phi\left(z^{(i)}\right)\right)\right)\right]+\frac{1}{2}\|\boldsymbol{w}\|^{2}$$

如果我们减小C的值，也就是增大正则系数的值，正则化项的威力也增强。

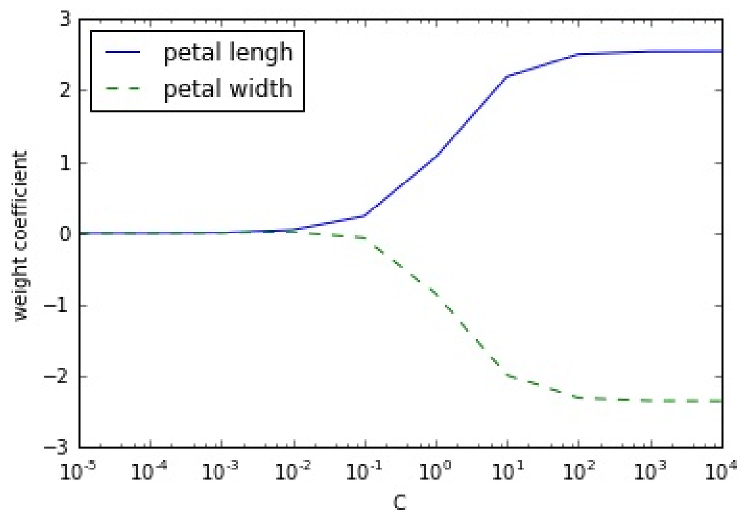
In [16]:
```
weights, params = [], []
```

In [17]:
```
for c in np.arange(-5, 5):
    lr = LogisticRegression(C=10**c, random_state=0)
    lr.fit(X_train_std, y_train)
    weights.append(lr.coef_[1])
    params.append(10**c)
```

In [19]:
```
weights = np.array(weights)
```

In [20]:
```
plt.plot(params, weights[:, 0],
         label='petal length')
plt.plot(params, weights[:, 1],
         linestyle='--',label='petal width')
plt.ylabel('weight coefficient')
plt.xlabel('C')
plt.legend(loc='upper left')
plt.xscale('log')
plt.show()
```

执行上面的代码，我们训练了十个带有不同C值的逻辑回归模型。

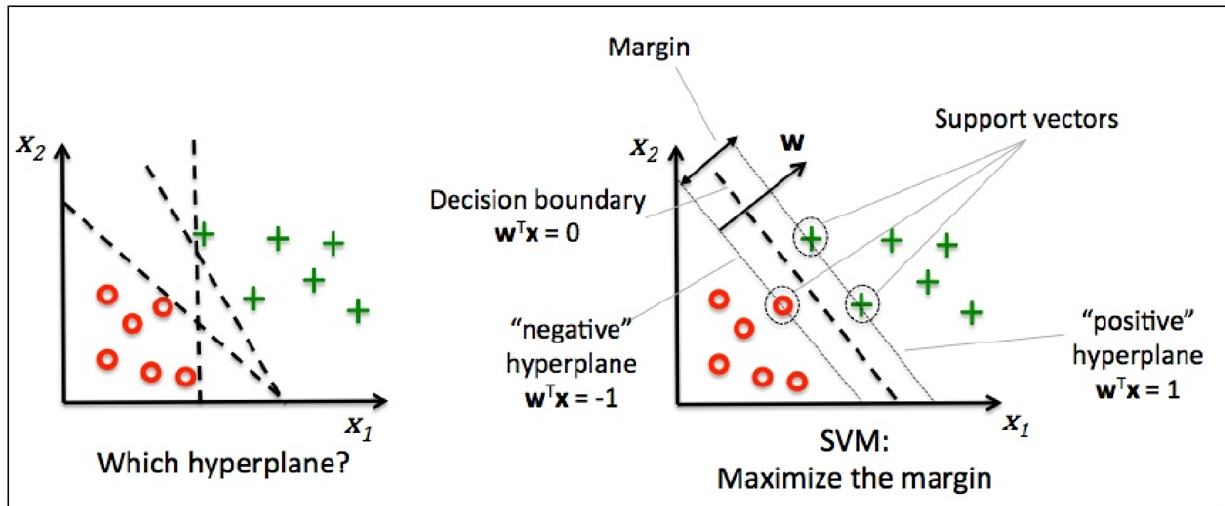


我们可以看到随着C的减小，权重系数也减小。

Note 本章只是简要介绍了逻辑斯蒂回归模型，如果你还意犹未尽，推荐阅读 Logistic Regression: From Introductory to Advanced Concepts and Applications, Sage Publications.

支持向量机

另一个经常使用的机器学习算法是支持向量机(support vector machine, SVM)，SVM可以看做是感知机的扩展。在感知机算法中，我们最小化错误分类误差。在SVM中，我们的优化目标是最大化间隔(margin)。间隔定义为两个分隔超平面(决策界)的距离，那些最靠近超平面的训练样本也被称为支持向量(suppor vectors)。可以看下图：



最大化间隔

最大化决策界的间隔，这么做的原因是间隔大的决策界趋向于含有更小的泛化误差，而间隔小的决策界更容易过拟合。为了更好地理解间隔最大化，我们先认识一下那些和决策界平行的正超平面和负超平面，他们可以表示为：

$$w_0 + \boldsymbol{w}^T \boldsymbol{x}_{pos} = 1 \qquad (1)$$

$$w_0 + \boldsymbol{w}^T \boldsymbol{x}_{neg} = -1 \qquad (2)$$

用(1)减去(2)，得到：

$$\Rightarrow \boldsymbol{w}^T \left(\boldsymbol{x}_{pos} - \boldsymbol{x}_{neg} \right) = 2$$

对上式进行归一化，

$$\frac{\mathbf{w}^T\left(\mathbf{x}_{pos}-\mathbf{x}_{neg}\right)}{\|\mathbf{w}\|}=\frac{2}{\|\mathbf{w}\|}$$

其中，$\|\mathbf{w}\|=\sqrt{\sum_{j=1}^{m} w_j^2}$。

上式等号左边可以解释为正超平面和负超平面之间的距离，也就是所谓的间隔。

现在SVM的目标函数变成了最大化间隔 ,限制条件是样本被正确分类，可以写成：

$$w_0+\mathbf{w}^T\mathbf{x}^{(i)}\geq 1 \ if \ y^{(i)}=1$$

$$w_0+\mathbf{w}^T\mathbf{x}^{(i)}<-1 \ if \ y^{(i)}=-1$$

上面两个限制条件说的是所有负样本要落在负超平面那一侧，所有正样本要落在正超平面那侧。我们用更简洁的写法代替：

$$y^{(i)}\left(w_0+\mathbf{w}^T\mathbf{x}^{(i)}\right)\geq 1 \ \forall_i$$

实际上，使用二次规划(quadratic programming)最小化 很容易，但是二次规划显然超出了本书的内容，如果你对SVM感兴趣，推荐阅读Vladimir Vapnik写的 The Nature of Statistical Learning Theory, Springer Science&Business Media或Chris J.C. Burges写很棒的解释A Tutorial on Support Vector Machines for Pattern Recognition.

使用松弛变量解决非线性可分的情况

虽然我们不想深挖SVM背后的数学概念，但还是有必要简短介绍一下松弛变量(slack variable) ,它是由Vladimir Vapnik在1995年引入的，借此提出了软间隔分类(soft-margin)。引入松弛变量的动机是原来的线性限制条件在面对非线性可分数据时需要松弛，这样才能保证算法收敛。

松弛变量值为正，添加到线性限制条件即可:

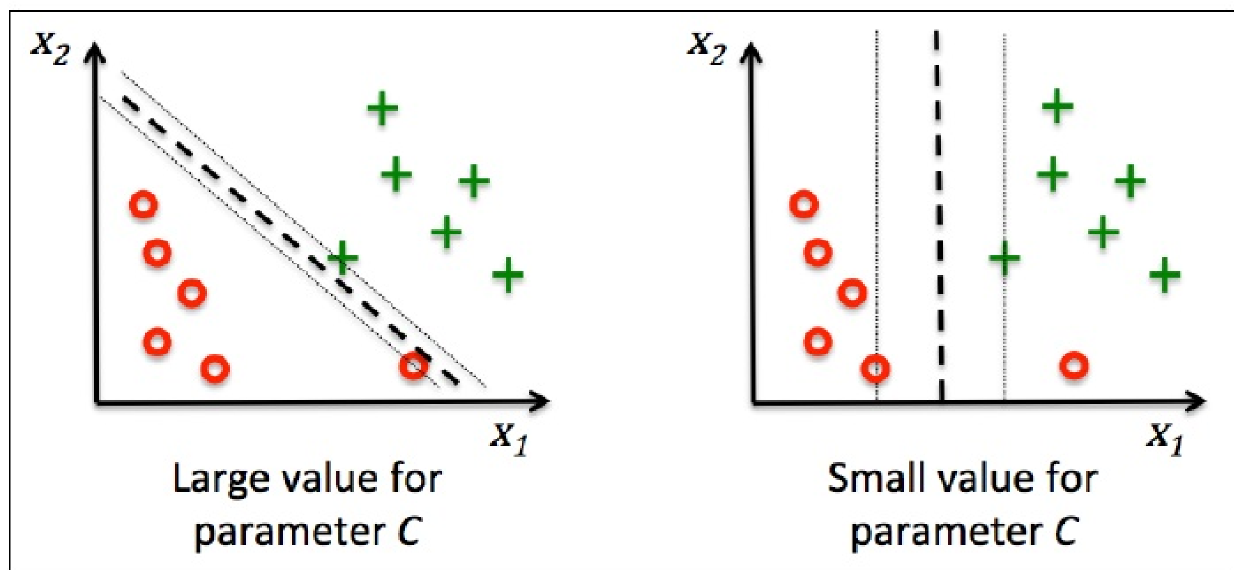
$$w^T x^{(i)} \geq 1 \ if \ \ y^{(i)} = 1 - \xi^{(i)}$$

$$w^T x^{(i)} < -1 \ if \ \ y^{(i)} = 1 + \xi^{(i)}$$

新的目标函数变成了:

$$\frac{1}{2}\|w\|^2 + C\left(\sum_i \xi^{(i)}\right)$$

使用变量C，我们可以控制错分类的惩罚量。和逻辑斯蒂回归不同，这里C越大，对于错分类的惩罚越大。可以通过C控制间隔的宽度，在bias-variance之间找到某种平衡：



这个概念和正则化相关，如果增大C的值会增加bias而减小模型的方差。

我们已经学会了线性SVM的基本概念，下面使用sklearn训练一个模型：

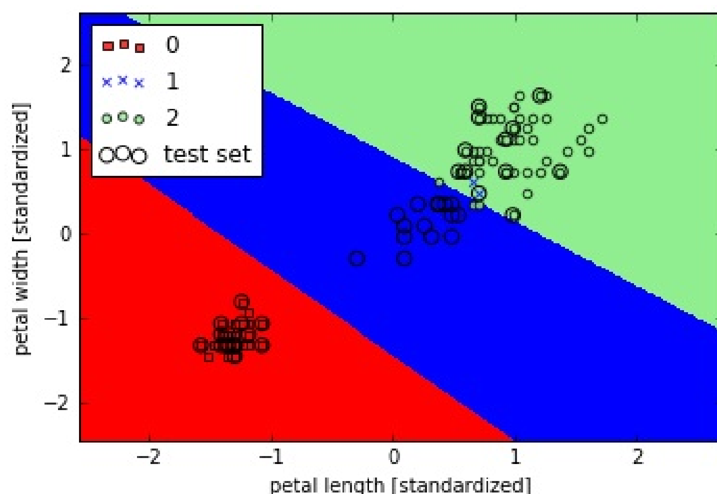
```
In [10]: from sklearn.svm import SVC

In [11]: svm = SVC(kernel='linear', C=1.0, random_state=0)

In [12]: svm.fit(X_train_std, y_train)

Out[12]: SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
           decision_function_shape=None, degree=3, gamma='auto', kernel='linear',
           max_iter=-1, probability=False, random_state=0, shrinking=True,
           tol=0.001, verbose=False)

In [13]: plot_decision_regions(X_combined_std, y_combined, classifier=svm,
                               test_idx=range(105,150))
         plt.xlabel('petal length [standardized]')
         plt.ylabel('petal width [standardized]')
         plt.legend(loc='upper left')
         plt.show()
```



Note 逻辑斯蒂回归 VS SVM

在解决现实的分类问题时，线性逻辑斯蒂回归和线性SVM通常效果近似。逻辑回归目标是最大化训练集的条件似然,使得她更易受奇异值影响。SVM只关心那些离决策界最近的点(即，支持向量)。另一方面，逻辑斯蒂回归的优点是易于实现，特别是并行化实现。此外，面对流式数据，逻辑斯蒂回归的易于更新的特点也很明显。

scikit-learn中不同的实现方式

前面我们用到的sklearn中的Perceptron和LogisticRegression类的实现都使用了LIBLINEAR库，LIBLINEAR是由国立台湾大学开发的一个高度优化过的C/C++库。同样，sklearn中的SVC类利用了国立台湾大学开发的LIBSVM库。

调用LIBLINEAR和LIBSVM而不是用Python自己实现的优点是训练模型速度很快，毕竟是优化过的代码。可是，有时候数据集很大，不能一次读入内存，针对这个问题，sklearn也实现了SGDClassifier类，使用提供的partial_fit方法能够支持在线学习。SGDClassifier利用随机梯度

下降算法学习参数。我们可以调用这一个类初始化随机梯度下降版本的感知机、逻辑斯蒂回归和SVM。

```
In [6]: from sklearn.linear_model import SGDClassifier

In [7]: ppn = SGDClassifier(loss='perceptron')

In [8]: lr = SGDClassifier(loss='log')

In [9]: svm=SGDClassifier(loss='hinge')
```

使用核SVM解决非线性

SVM之所以受欢迎度这么高，另一个重要的原因是它很容易核化(kernelized)，能够解决非线性分类问题。在讨论核SVM细节之前，我们先自己创造一个非线性数据集，看看他长什么样子。

使用下面的代码，我们将创造一个简单的数据集，其中100个样本是正类，100个样本是负类。

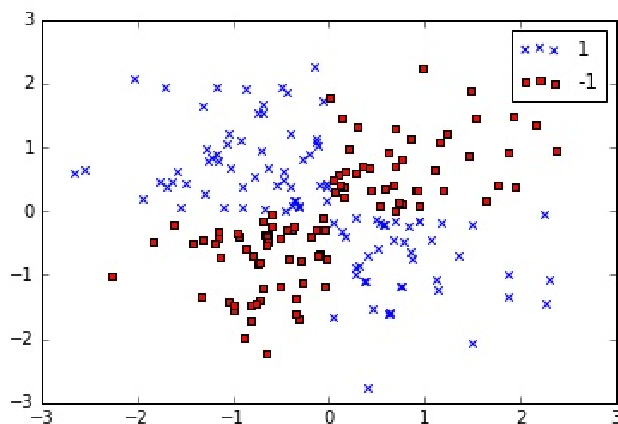
In [10]:
```
np.random.seed(0)
```

In [11]:
```
X_xor = np.random.randn(200, 2)
```

In [12]:
```
y_xor = np.logical_xor(X_xor[:, 0] > 0, X_xor[:, 1] > 0)
```

In [13]:
```
y_xor = np.where(y_xor, 1, -1)
```

In [16]:
```
plt.scatter(X_xor[y_xor == 1, 0], X_xor[y_xor ==1, 1],
            c='b', marker='x', label='1')
plt.scatter(X_xor[y_xor == -1, 0], X_xor[y_xor==-1, 1],
            c='r', marker='s', label='-1')
plt.ylim(-3.0)
plt.legend()
plt.show()
```



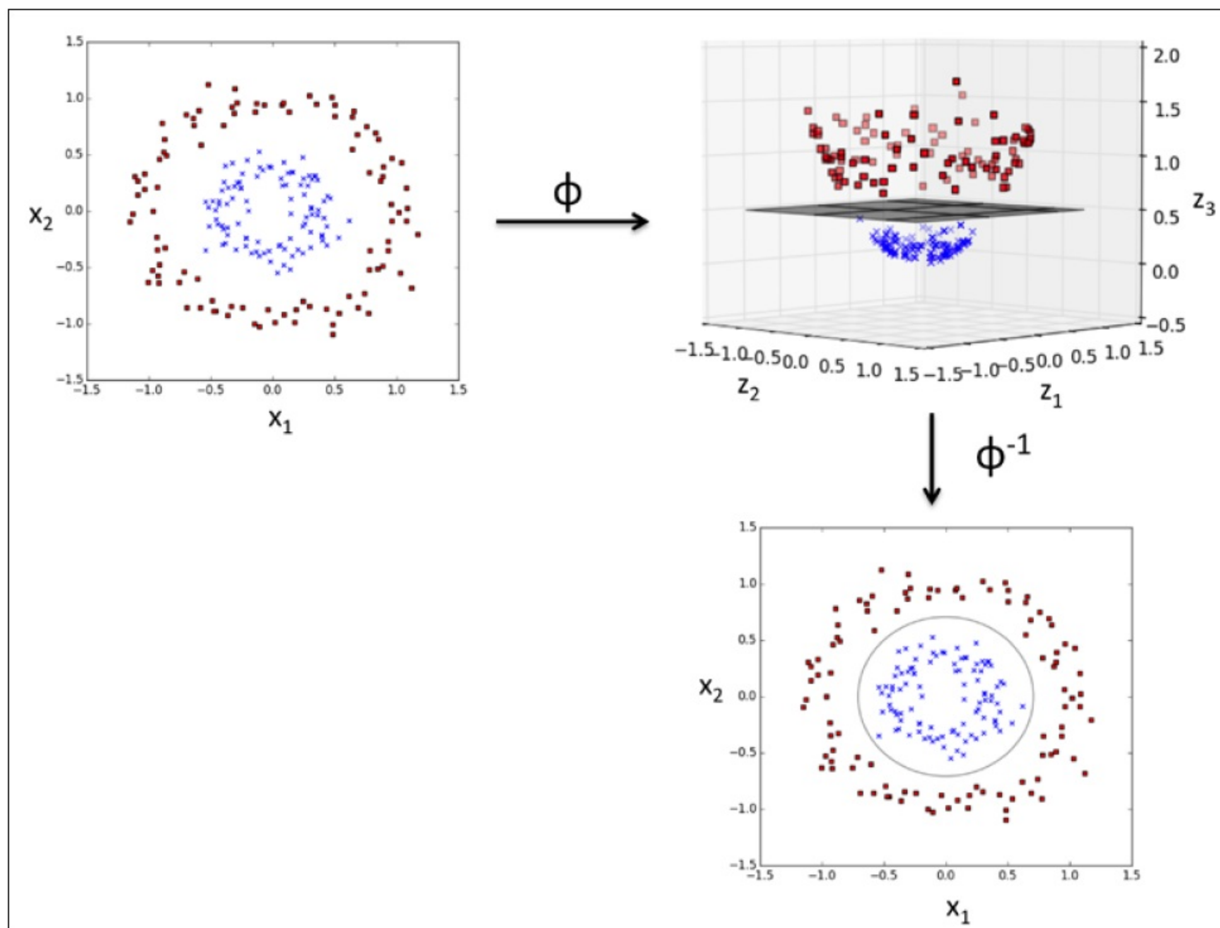
显然，如果要用线性超平面将正负类分开是不可能的，所以前面介绍的线性逻辑斯蒂回归和线性SVM都鞭长莫及。

核方法的idea是为了解决线性不可分数据，在原来特征基础上创造出非线性的组合，然后利用映射函数 将现有特征维度映射到更高维的特征空间，并且这个高维度特征空间能够使得原来线性不可分数据变成了线性可分的。

举个例子，下图中，我们将两维的数据映射到三位特征空间，数据集也有线性不可分变成了线性可分，使用的映射为：

$$\phi(x_1,x_2)=(z_1,z_2,z_3)=(x_1,x_2,x_1^2+x_2^2)$$

注意看右上角子图到右下角子图的转变，高维空间中的线性决策界实际上是低维空间的非线性决策界，这个非线性决策界是线性分类器找不到的，而核方法找到了：



使用核技巧在高维空间找到可分超平面

使用SVM解决非线性问题，我们通过映射函数 将训练集映射到高维特征空间，然后训练一个线性SVM模型在新特征空间将数据分类。然后，我们可以使用相同的映射函数对测试集数据分类。

上面的想法很不错，但是如何构建新特征是非常困难的，尤其是数据本身就是高维数据时。因此，我们就要介绍核技巧了。由于我们不会过多涉及在训练SVM时如何求解二次规划问题，你只需要知道用 替换$x^{(i)T}x^{(j)}$就可以了。为了免去两个点的点乘计算， 我们定义所谓

$$k\left(x^{(i)},x^{(j)}\right)=\phi\left(x^{(i)}\right)^T\phi\left(x^{(j)}\right).$$

的核函数(kernel function):

常用的一个核函数是Radial Basis Function kernel(RBF核)，也称为高斯核:

$$k\left(x^{(i)}, x^{(j)}\right) = \exp\left(-\frac{\left\|x^{(i)} - x^{(j)}\right\|^2}{2\sigma^2}\right)$$

通常简写为：

$$k\left(x^{(i)}, x^{(j)}\right) = \exp\left(-\gamma\left\|x^{(i)} - x^{(j)}\right\|^2\right)$$

此处，,是一个要优化的自由参数。

通俗地讲，核(kernel)可以被解释为两个样本之间的相似形函数。高斯核中e的指数范围<=0,因此高斯核值域范围,特别地，当两个样本完全一样时，值为1，两个样本完全不同时，值为0.

有了核函数的概念，我们就动手训练一个核SVM，看看是否能够对线性不可分数据集正确分类：

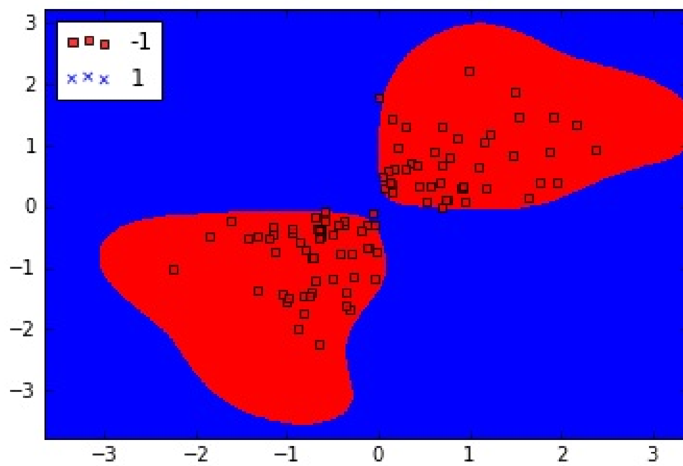
In [13]:
```
svm = SVC(kernel='rbf', random_state=0, gamma=1.0, C=10.0)
```

In [14]:
```
svm.fit(X_xor, y_xor)
```

Out[14]:
```
SVC(C=10.0, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape=None, degree=3, gamma=1.0, kernel='rbf',
  max_iter=-1, probability=False, random_state=0, shrinking=True,
  tol=0.001, verbose=False)
```

In [15]:
```
plot_decision_regions(X_xor, y_xor, classifier=svm)
plt.legend(loc='upper left')
plt.show()
```

结果如下，可以发现核SVM在XOR数据集上表现相当不错：



其中参数gamma可以被理解为高斯球面的阶段参数，如果我们增大gamma值，会产生更加柔软的决策界。为了更好地理解gamma参数，我们在Iris数据集上应用RBF核SVM：

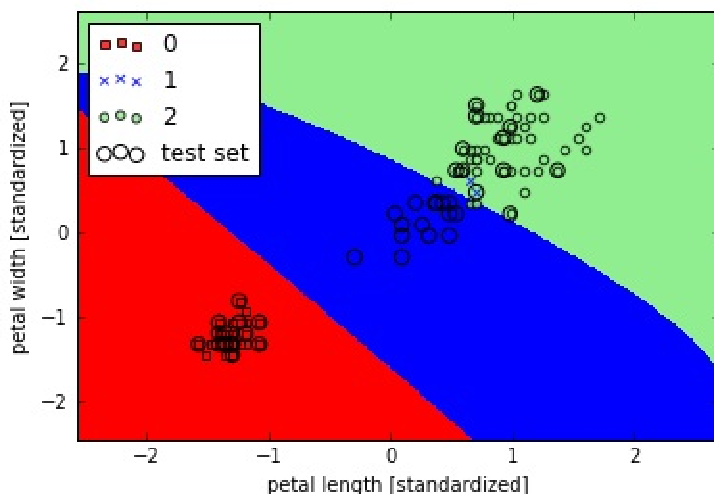
```
In [16]: svm = SVC(kernel='rbf', random_state=0, gamma=0.2, C=1.0)

In [17]: svm.fit(X_train_std, y_train)

Out[17]: SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
           decision_function_shape=None, degree=3, gamma=0.2, kernel='rbf',
           max_iter=-1, probability=False, random_state=0, shrinking=True,
           tol=0.001, verbose=False)

In [22]: plot_decision_regions(X_combined_std, y_combined, classifier=svm, test_idx=range(105, 150))
         plt.xlabel('petal length [standardized]')
         plt.ylabel('petal width [standardized]')
         plt.legend(loc='upper left')
         plt.show()
```

我们选择的gamma值相对比较小，所以决策界比较soft：



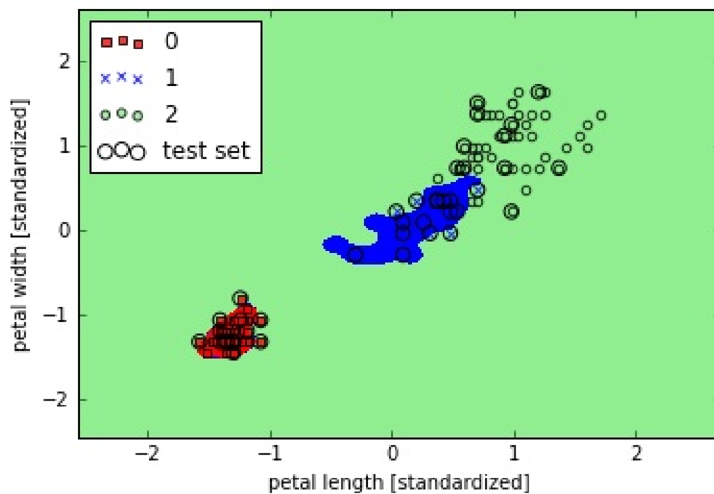
现在我们增大gamma值，然后观察决策界：

```
In  [23]: svm = SVC(kernel='rbf', random_state=0, gamma=100, C=1.0)

In  [24]: svm.fit(X_train_std, y_train)

Out[24]: SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
           decision_function_shape=None, degree=3, gamma=100, kernel='rbf',
           max_iter=-1, probability=False, random_state=0, shrinking=True,
           tol=0.001, verbose=False)

In  [25]: plot_decision_regions(X_combined_std, y_combined, classifier=svm, test_idx=range(105, 150))
          plt.xlabel('petal length [standardized]')
          plt.ylabel('petal width [standardized]')
          plt.legend(loc='upper left')
          plt.show()
```

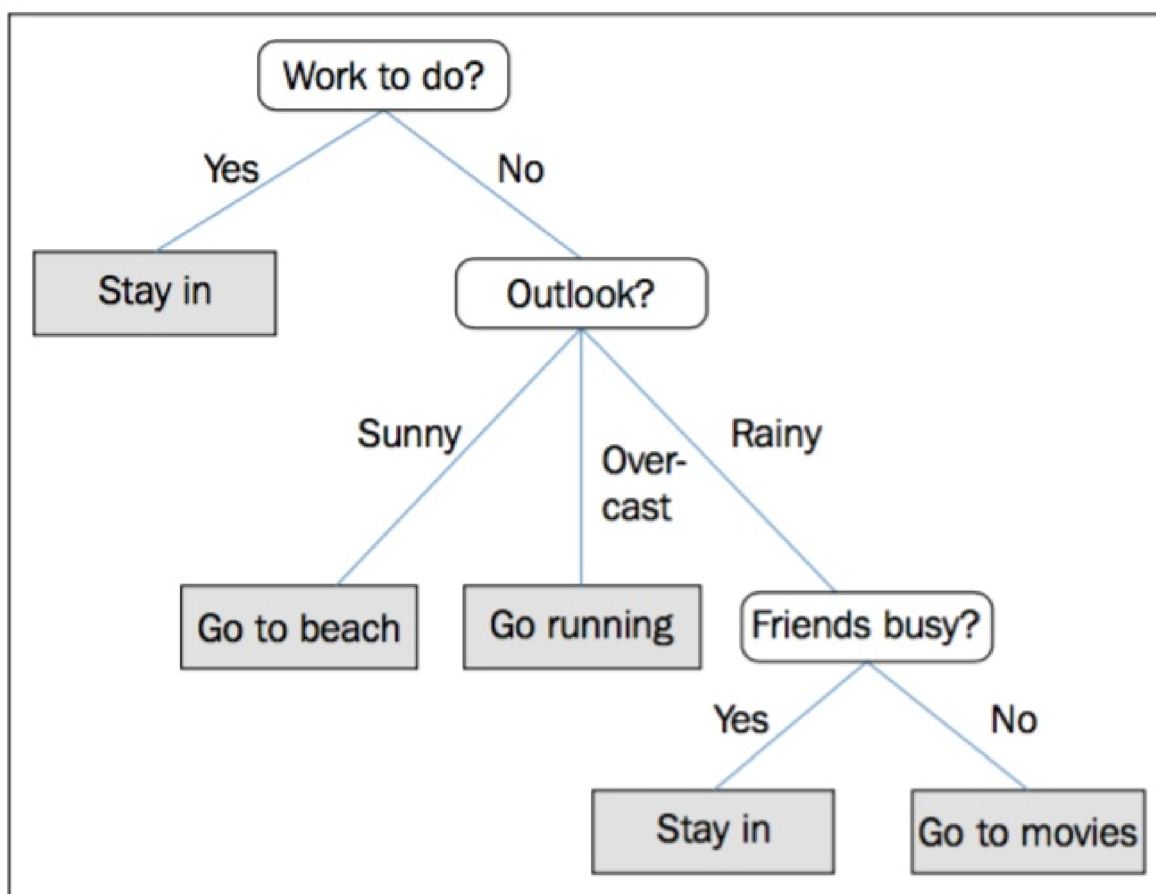


虽然gamma值较大的模型对训练集分类效果很大，但其泛化能力一般很差，所以选择适当的gamma值有助于避免过拟合。

决策树学习

如果我们在意模型的可解释性，那么决策树(decision tree)分类器绝对是上佳的选择。如同名字的字面意思，我们可以把决策树理解为基于一系列问题对数据做出的分割选择。

举一个简单的例子，我们使用决策树来决定某一天的活动：



基于训练集中的特征，决策树模型提出了一系列问题来推测样本的类别。虽然上图中做出的每个决策都是根据离散变量，但也可以用于连续型变量，比如，对于Iris中sepal width这一取值为实数的特征，我们可以问“sepal width是否大于2.8cm？”

训练决策树模型时，我们从根节点出发，使用信息增益(information gain, IG)最大的特征对数据分割。然后迭代此过程。显然，决策树的生成是一个递归过程，在决策树基本算法中，有三种情形会导致递归返回:（1）当前节点包含的样本全属于同一类别，无需划分；（2）当前属性集为空，或是所有样本在所有属性上取值相同，无法划分；（3）当前节点包含的样本集合为空，不能划分。

每一个节点的样本都属于同一个类，同时这也可能导致树的深度很大，节点很多，很容易引起过拟合。因此，剪枝操作是必不可少的，来控制树深度。

最大信息增益

为了使用最大信息增益的特征分割数据，我们需要定义一个在决策树学习过程中的目标函数。此处，我们的目标函数是在每一次分割时最大化信息增益，我们定义如下：

$$IG(D_p, f) = I(D_p) - \sum_{j=1}^{m} \frac{N_j}{N_p} I(D_j)$$

其中，f是具体的特征，$D_p D_j$是当前数据集和用特征f分割后第j个子节点的数据集，I是某种度量，N_p是当前数据集样本个数，N_j是第j个子节点的数据集中样本个数。为了简化和减小搜索空间，大多数决策树(包括sklearn)都是用二叉树实现的。这意味着每一个父节点被分割为两个子节点，$D_{left} D_{right}$:

$$IG(D_p, a) = I(D_p) - \frac{N_{left}}{N_p} I(D_{left}) - \frac{N_{right}}{N_p} I(D_{right})$$

常用的度量I包括基尼指数(Gini index, I_G)、熵(Entropy, I_H)和分类错误(classification error, I_E)。我们以熵为例：

$$I_H(t) = -\sum_{i=1}^{c} p(i|t) \log_2 p(i|t)$$

此处,$p(i|t)$指的是在节点t中属于类别c的样本占比。所以如果某个节点中所有样本都属于同一个类，则熵为0，如果样本的类别时均匀分布，则熵最大。比如，在二分类情况下，如果p(i=1|t)=1或p(i=0|t)=0，则熵为0.因此，我们说熵的评价标准目的是最大化树中的互信息。

基尼系数可以被理解为最小化误分类的概率：

$$I_G(t) = \sum_{i=1}^{c} p(i|t)(-p(i|t)) = 1 - \sum_{i=1}^{c} p(i|t)^2$$

和熵一样，如果节点中样本的类别均匀，则基尼系数最大，比如，在二分类情况下：

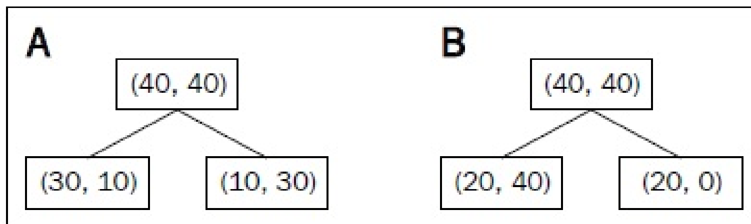
$$1-\sum_{i=1}^{c} 0.5^2 = 0.5$$

通常，熵和基尼系数的结果相似，所以不需要花太多时间在选择度量上面。

另一种常用的度量是分类误差：

$$I_E = 1 - \max\left\{p\left(i \mid t\right)\right\}$$

这个度量更建议在剪枝时使用，而不是在构建决策树时使用。举个简单的例子，说明一下为什么不建议构建树时用：



D_p中包含40个正例样本和40个负例样本，然后分割为两个子节点。信息增益使用分类误差作为度量，得到的值在A、B情况下相同，都是0.25,计算过程如下：

$$I_E\left(D_p\right)=1-0.5=0.5$$

$$A:I_E\left(D_{left}\right)=1-\frac{3}{4}=0.25$$

$$A:I_E\left(D_{right}\right)=1-\frac{3}{4}=0.25$$

$$A:IG_E=0.5-\frac{4}{8}0.25-\frac{4}{8}0.25=0.25$$

$$B:I_E\left(D_{left}\right)=1-\frac{4}{6}=\frac{1}{3}$$

$$B:I_E\left(D_{right}\right)=1-1=0$$

$$B:IG_E=0.5-\frac{6}{8}\times\frac{1}{3}-0=0.25$$

如果使用基尼系数，则会按照B情况分割：

$$I_G\left(D_p\right)=1-\left(0.5^2+0.5^2\right)=0.5$$

$$A:I_G\left(D_{left}\right)=1-\left(\left(\frac{3}{4}\right)^2+\left(\frac{1}{4}\right)^2\right)=\frac{3}{8}=0.375$$

$$A:I_G\left(D_{right}\right)=1-\left(\left(\frac{1}{4}\right)^2+\left(\frac{3}{4}\right)^2\right)=\frac{3}{8}=0.375$$

$$A:I_G=0.5-\frac{4}{8}0.375-\frac{4}{8}0.375=0.125$$

$$B:I_G\left(D_{left}\right)=1-\left(\left(\frac{2}{6}\right)^2+\left(\frac{4}{6}\right)^2\right)=\frac{4}{9}=0.\overline{4}$$

$$B:I_G\left(D_{right}\right)=1-\left(1^2+0^2\right)=0$$

$$B:IG_G=0.5-\frac{6}{8}0.\overline{4}-0=0.1\overline{\overline{6}}$$

同样，如果用熵作为度量，也会按照B分割：

$$I_H\left(D_p\right)=-\left(0.5\ \log_2\left(0.5\right)+0.5\ \log_2\left(0.5\right)\right)=1$$

$$A:I_H\left(D_{left}\right)=-\left(\frac{3}{4}\log_2\left(\frac{3}{4}\right)+\frac{1}{4}\log_2\left(\frac{1}{4}\right)\right)=0.81$$

$$A: I_H\left(D_{right}\right) = -\left(\frac{1}{4}\log_2\left(\frac{1}{4}\right) + \frac{3}{4}\log_2\left(\frac{3}{4}\right)\right) = 0.81$$

$$A: IG_H = 1 - \frac{4}{8}0.81 - \frac{4}{8}0.81 = 0.19$$

$$B: I_H\left(D_{left}\right) = -\left(\frac{2}{6}\log_2 + \left(\frac{2}{6}\right) + \frac{4}{6}\log_2 + \left(\frac{4}{6}\right)\right) = 0.92$$

$$B: I_H\left(D_{right}\right) = 0$$

$$B: IG_H = 1 - \frac{6}{8}0.92 - 0 = 0.31$$

构建一棵决策树

决策树通过将特征空间分割为矩形，所以其决策界很复杂。但是要知道过大的树深度会导致过拟合，所以决策界并不是越复杂越好。我们调用sklearn，使用熵作为度量，训练一颗最大深度为3的决策树。还有一点，对于决策树算法来说，特征缩放并不是必须的。代码如下：

```
In [26]: from sklearn.tree import DecisionTreeClassifier
```

```
In [27]: tree = DecisionTreeClassifier(criterion='entropy', max_depth=3, random_state=0)
```

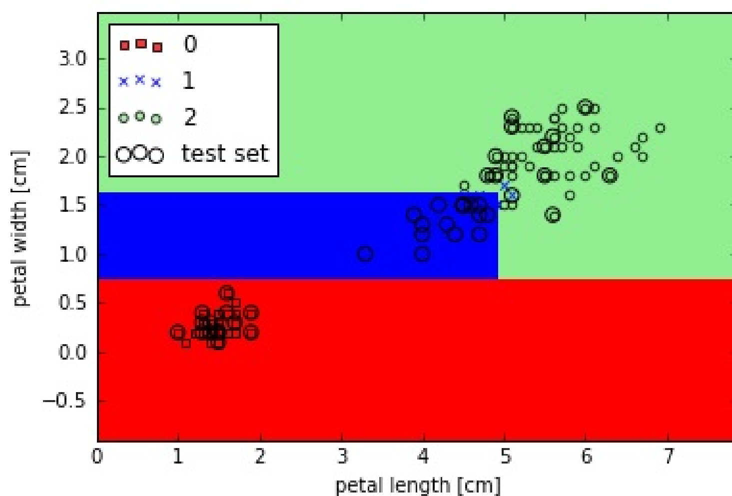
```
In [28]: tree.fit(X_train, y_train) # NO standardized
```

```
Out[28]: DecisionTreeClassifier(class_weight=None, criterion='entropy', max_depth=3,
                 max_features=None, max_leaf_nodes=None, min_samples_leaf=1,
                 min_samples_split=2, min_weight_fraction_leaf=0.0,
                 presort=False, random_state=0, splitter='best')
```

```
In [29]: X_combined = np.vstack((X_train, X_test))
         y_combined = np.hstack((y_train, y_test))
```

```
In [30]: plot_decision_regions(X_combined, y_combined, classifier=tree, test_idx=range(105, 150))
         plt.xlabel('petal length [cm]')
         plt.ylabel('petal width [cm]')
         plt.legend(loc='upper left')
         plt.show()
```

执行上面的代码，我们得到如下结果，决策界和坐标轴平行：



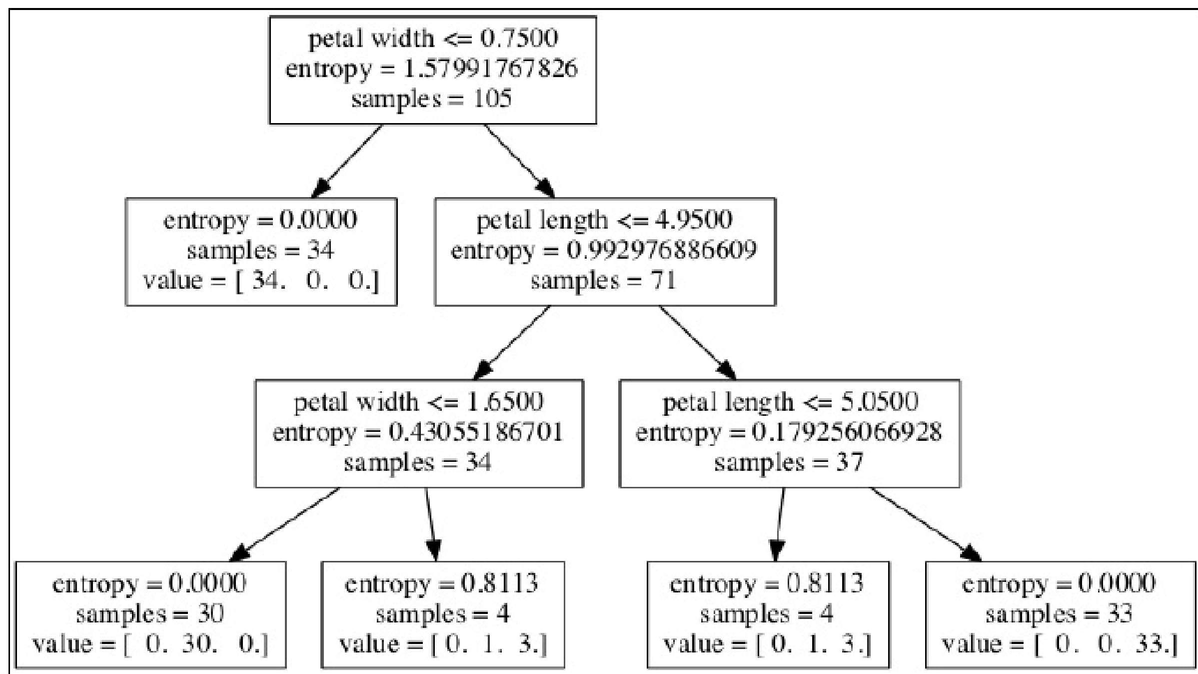
sklearn的一大优点是可以将训练好的决策树模型输出，保存在.dot文件，我们可以利用GraphViz对其可视化。

先调用sklearn中export_graphviz将树模型导出:

```
In [31]: from sklearn.tree import export_graphviz
```

```
In [32]: export_graphviz(tree, out_file='tree.dot', feature_names=['petal length', 'petal width'])
```

然后利用GraphViz程序将tree.dot转为PNG图片:



现在我们可以查看决策树在构建树时的过程：根节点105个样本，使用 petal_width <=0.75分割为两个子节点。经过第一个分割，我们可以发现左节点中样本都是同一类型，所以停止此节点的分割，右节点继续分割，注意一点，在构建决策树时两个特征各使用了两次。

随机森林

随机森林一直是广受欢迎的模型，优点很多：优秀的分类表现、扩展性和使用简单。随机森林的思想也不复杂，一个随机森林模型就是多颗决策树的集成。集成学习(ensemble learning)的观点是将多个弱分类器结合来构建一个强分类器，它的泛化误差小且不易过拟合。

随机森林算法大致分为4个步骤：

- 通过自助法(bootstrap)构建大小为n的一个训练集，即重复抽样选择n个训练样例
- 对于刚才新得到的训练集，构建一棵决策树。在每个节点执行以下操作：
-
    - 通过不重复抽样选择d个特征
-
    - 利用上面的d个特征，选择某种度量分割节点
- 重复步骤1和2，k次
- 对于每一个测试样例，对k颗决策树的预测结果进行投票。票数最多的结果就是随机森林的预测结果。至于如何投票，下面会讲到。

随机森林中构建决策树的做法和原始决策树的区别是，在每次分割节点时，不是从所有特征中选择而是在一个小特征集中选择特征。

虽然随机森林模型的可解释性不如决策树，但是它的一大优点是受超参数的影响波动不是很大(译者注：几个主要参数还是需要好好调参的)。我们也不需要对随机森林进行剪枝因为集成模型的鲁棒性很强，不会过多受单棵决策树噪音的影响。

在实际运用随机森林模型时，树的数目(k)需要好好调参。一般，k越大，随机森林的性能越好，当然计算成本也越高。

样本大小n能够控制bias-variance平衡，如果n很大，我们就减小了随机性因此随机森林就容易过拟合。另一方面，如果n很小，虽然不会过拟合，但模型的性能会降低。大多数随机森林的实现，包括sklearn中的RandomForestClassifier，n的大小等于原始训练集的大小。

在每一次分割时特征集的大小d，一个最起码的要求是要小于原始特征集大小，sklearn中的默认值$d = m^{0.5}$，其中m是原始特征集大小，这是一个比较合理的数值。

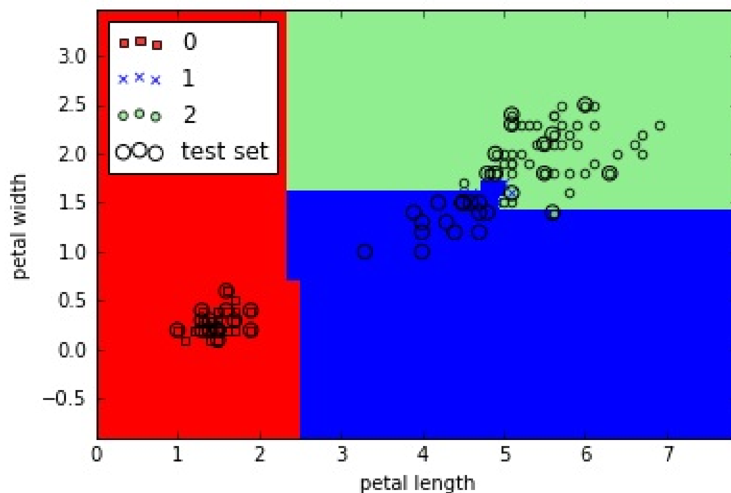
直接调用sklearn来看一下随机森林吧：

```
In [33]: from sklearn.ensemble import RandomForestClassifier
         forest = RandomForestClassifier(criterion='entropy', n_estimators=10, random_state=1,n_jobs=2)
```

```
In [34]: forest.fit(X_train, y_train)
```

```
Out[34]: RandomForestClassifier(bootstrap=True, class_weight=None, criterion='entropy',
                     max_depth=None, max_features='auto', max_leaf_nodes=None,
                     min_samples_leaf=1, min_samples_split=2,
                     min_weight_fraction_leaf=0.0, n_estimators=10, n_jobs=2,
                     oob_score=False, random_state=1, verbose=0, warm_start=False)
```

```
In [35]: plot_decision_regions(X_combined, y_combined, classifier=forest, test_idx=range(105, 150))
         plt.xlabel('petal length')
         plt.ylabel('petal width')
         plt.legend(loc='upper left')
         plt.show()
```



运行上面的代码，我们训练了一个含有10颗树的随机森林，使用熵作为分割节点时的度量。虽然我们在一个小数据集上训练了一个非常小的模型，但我还是使用了n_jobs这个并行化参数，此处使用了计算机的两个核训练模型。

k近邻--一个懒惰学习算法

本章我们要讨论的最后一个监督学习算法是k紧邻算法(k-nearest neighbor classifier, KNN), 这个算法很有意思，因为他背后的思想和本章其他算法完全不同。

KNN是懒惰学习的一个典型示例。之所以称为“懒惰”并不是由于此类算法看起来很简单，而是在训练模型过程中这类算法并不去学习一个判别式函数(损失函数)而是要记住整个训练集。

Note 参数模型VS变参模型

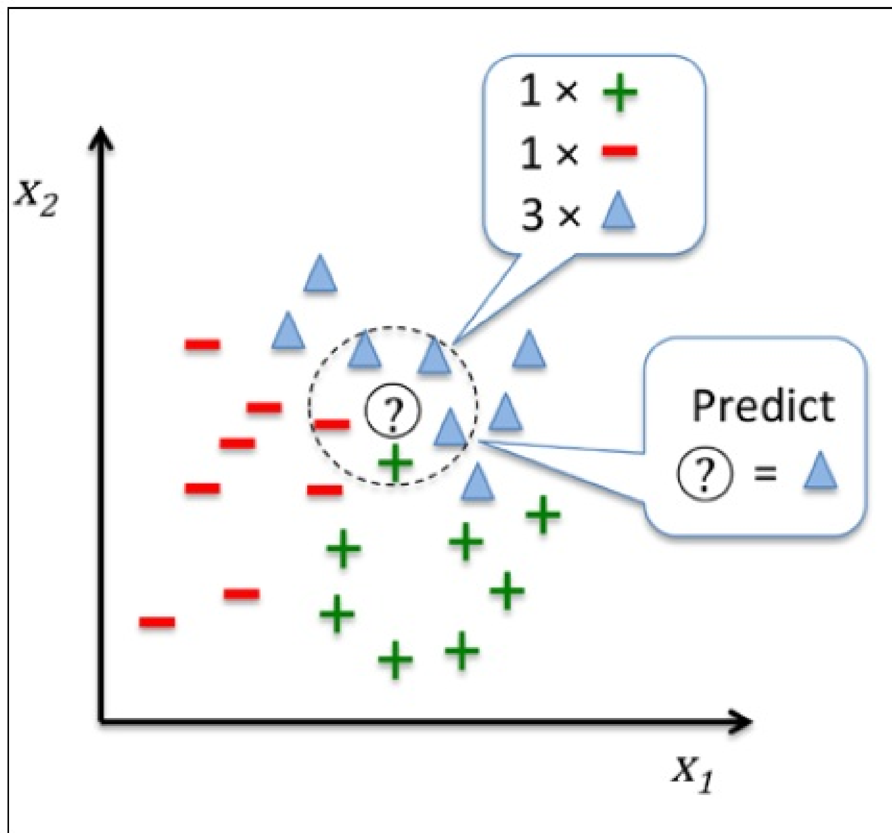
机器学习算法可以被分为两大类：参数模型和变参模型。对于参数模型，在训练过程中我们要学习一个函数，重点是估计函数的参数，然后对于新数据集，我们直接用学习到的函数对齐分类。典型的参数模型包括感知机、逻辑斯蒂回归和线性SVM。与之相对的，变参模型中的参数个数不是固定的，它的参数个数随着训练集增大而增多！很多书中变参(nonparametric)被翻译为无参模型，一定要记住，不是没有参数，而是参数个数是变量！变参模型的两个典型示例是决策树/随机森林和核SVM。

KNN属于变参模型的一个子类：基于实例的学习(instance-based learning)。基于实例的学习的模型在训练过程中要做的是记住整个训练集，而懒惰学习是基于实例的学习的特例，在整个学习过程中不涉及损失函数的概念。

KNN算法本身非常简单，步骤如下：

- 1 确定k大小和距离度量。
- 2 对于测试集中的一个样本，找到训练集中和它最近的k个样本。
- 3 将这k个样本的投票结果作为测试样本的类别。

一图胜千言，请看下图：



对每一个测试样本，基于事先选择的距离度量，KNN算法在训练集中找到距离最近(最相似)的k个样本，然后将k个样本的类别的投票结果作为测试样本的类别。

像KNN这种基于内存的方法一大优点是：一旦训练集增加了新数据，模型能立刻改变。另一方面，缺点是分类时的最坏计算复杂度随着训练集增大而线性增加，除非特征维度非常低并且算法用诸如KD-树等数据结构实现。此外，我们要一直保存着训练集，不像参数模型训练好模型后，可以丢弃训练集。因此，存储空间也成为了KNN处理大数据的一个瓶颈。

下面我们调用sklearn训练一个KNN模型：

```
In [10]: from sklearn.neighbors import KNeighborsClassifier
```

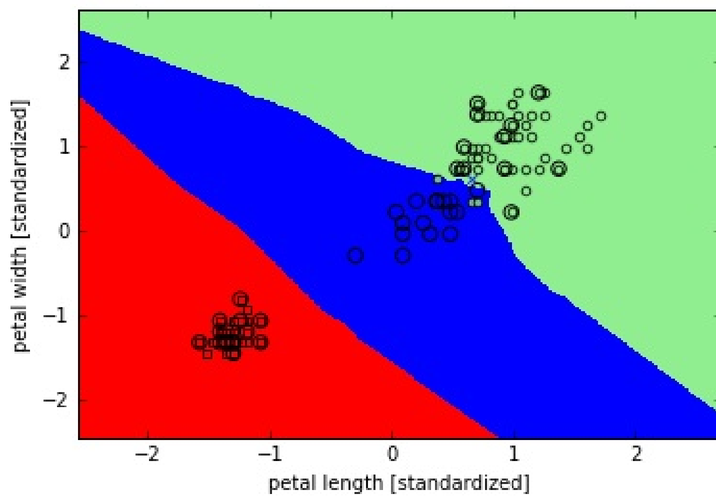
```
In [11]: knn = KNeighborsClassifier(n_neighbors=5, p=2, metric='minkowski')
```

```
In [12]: knn.fit(X_train_std, y_train)

Out[12]: KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                    metric_params=None, n_jobs=1, n_neighbors=5, p=2,
                    weights='uniform')
```

```
In [13]: plot_decision_regions(X_combined_std, y_combined, classifier=knn, test_idx=range(105, 150))
         plt.xlabel('petal length [standardized]')
         plt.ylabel('petal width [standardized]')
         plt.show()
```

我们设置k=5，得到了相对平滑的决策界：



k的选择对于KNN模型来说至关重要，除此之外，距离度量也是很有用的。通常，欧氏距离用于实数域的数据集，此时一定要对特征进行标准化，这样每一维度特征的重要性等同。我们在上面的代码中使用的距离度量是'minkowski',它是欧氏距离和曼哈顿距离的一般化：

$$d\left(\boldsymbol{x}^{(i)},\boldsymbol{x}^{(i)}\right)=\sqrt[p]{\sum_k\left|x_k^{(i)}\ x_k^{(j)}\right|^p}$$

如果p=2，则退化为欧氏距离，p=1，则退化为曼哈顿距离。使用metric参数可以选择不同的距离度量。

Note 维度诅咒

注意，如果特征维度过大，KNN算法很容易过拟合。我们可以想象，对于一个固定大小的训练集，如果特征空间维度非常高，空间中最相似的两个点也可能距离很远(差别很大)。虽然我们在逻辑回归中讨论了正则项来防止过拟合，但是正则项却不适用于KNN和决策树模型，我们只能通过特征选择和降维手段来避免维度诅咒。下一章我们会讲到。

总结

本章，你学习了许多不同的机器学习算法，用于解决线性和非线性问题。如果我们关注模型可解释性，决策树是很好的选择。逻辑斯蒂回归不但可以在在线学习场景下大展拳脚还能预测概率。虽然SVM能解决线性和非线性问题，但是它参数个数比较多，调参挺麻烦。集成算法包括随机森林则不需要调节过多的参数，也不会像决策树一样容易过拟合，这使得它很受欢迎。KNN通过懒惰学习进行分类，他不需要模型训练的过程但是在预测时的计算成本相对比较高。

然而，比选择合适的算法更重要的是训练集数据本身。如果数据的特征不够好，再好的算法也没用。

在下一章，我们会讨论预处理涉及到的内容。

第四章 构建一个好的训练集---数据预处理

数据的质量和包含的有用信息量是决定一个机器学习算法能够学多好的关键因素。因此，我们在训练模型前评估和预处理数据显得至关重要。在本章，我们要讨论必不可少的预处理技术，能够帮助我们构建更好的机器学习模型。

本章涉及的主题：

- 移除数据集中的缺失值
- 将分类(category)数据转型，能够被机器学习算法处理
- 特征选择

处理缺失值

现实中的数据总是存在或多或少的缺失值现象。原因多种多样，可能是数据收集阶段发生错误，也可能数据调研阶段某些选项没有被填写。不论是什么原因造成的缺失值，我们都统一将其看做空格或者用NaN(Not a Number)表示的占位符。

不幸地是，大多数计算工具不能处理缺失值，即使我们忽略缺失值也不能产生预测结果。因此，我们必须认真对待缺失值问题。在讨论处理缺失值的方法之前，我们先创建一个例子，以便更好地理解缺失值问题：

In [2]:
```
from io import StringIO
```

In [3]:
```
csv_data = '''A,B,C,d
              1.0,2.0,3.0,4.0
              5.0,6.0,,8.0
              0.0,11.0,12.0,'''
```

In [4]:
```
csv_data = unicode(csv_data)
```

In [5]:
```
df = pd.read_csv(StringIO(csv_data))
```

In [6]:
```
df
```

Out[6]:

	A	B	C	d
0	1.0	2.0	3.0	4.0
1	5.0	6.0	NaN	8.0
2	0.0	11.0	12.0	NaN

上面的代码，我们创建了一个csv格式的变量csv_data, 然后读入DataFrame对象，注意其中两个缺失值被NaN替代了。

如果DataFrame对象包含的数据很多，人工来查找NaN就不现实了。我们可以使用isnull方法来返回一个值为布尔类型的DataFrame，判断每个元素是否缺失，如果元素缺失，值为True。然后使用sum方法，我们就能得到DataFrame中每一列的缺失值个数，还是看代码理解吧：

In [7]:
```
df.isnull().sum()
```

Out[7]:
```
A    0
B    0
C    1
d    1
dtype: int64
```

现在我们知道如果统计DataFrame每一列中的缺失值个数。下一节我们学习几种处理缺失值的策略。

Note 虽然scikit-learn和NumPy数组结合的很方便，但是预处理时还是推荐使用pandas的DataFrame格式而非NumPy数组。由DataFrame对象得到NumPy数组很方便，直接通过values属性即可，然后就可以用sklearn中的算法了：

```
In  [8]: df.values

Out[8]: array([[  1.,   2.,   3.,   4.],
               [  5.,   6.,  nan,   8.],
               [  0.,  11.,  12.,  nan]])
```

消除带有缺失值的特征或样本

处理缺失值最简单的手段无疑是直接将带有缺失值的特征(列)或样本(行)从数据集中去掉。去掉行可以通过dropna方法：

```
In [9]: df.dropna()
```

Out[9]:

	A	B	C	d
0	1.0	2.0	3.0	4.0

去掉列同样用dropna方法，只不过将参数axis设置为1：

```
In [10]: df.dropna(axis=1)
```

Out[10]:

	A	B
0	1.0	2.0
1	5.0	6.0
2	0.0	11.0

dropna方法包含多个参数：

```
In [11]: #只去掉那些所有值均为NaN的行
         df.dropna(how='all')
```

Out[11]:

	A	B	C	d
0	1.0	2.0	3.0	4.0
1	5.0	6.0	NaN	8.0
2	0.0	11.0	12.0	NaN

```
In [12]: # 去掉那些非缺失值小于4个的行
         df.dropna(thresh=4)
```

Out[12]:

	A	B	C	d
0	1.0	2.0	3.0	4.0

In [13]:
```
# 去掉那些在特定列出现NaN的行
df.dropna(subset=['C'])
```

Out[13]:

	A	B	C	d
0	1.0	2.0	3.0	4.0
2	0.0	11.0	12.0	NaN

虽然移除缺失值的操作很简单，但是这样做的缺点也很明显，比如，如果每一行都有一个缺失值，那整个数据集都被移除了。或者移除了过多的特征。

下一节我们学习处理缺失值真正常用的做法：插入法(interpolation)。

改写缺失值

用移除法处理缺失值的缺点很明显，过于暴力，会丢掉很多数据信息。怎样保留那些非缺失值数据的同时处理缺失值呢？这就是插入法，用一个估计值来替代缺失值。最常用的是平均估计法，即用整个特征列的平均值代替这一列的缺失值。

使用sklearn中的Imputer类能很容易实现此方法：

```
In [19]: from sklearn.preprocessing import Imputer
```

```
In [15]: imr = Imputer(missing_values='NaN', strategy='mean', axis=0)
         imr = imr.fit(df)
```

```
In [16]: imputed_data = imr.transform(df.values)
```

```
In [24]: print df.values
         print('---------------------------------------------')
         print imputed_data

[[  1.    2.    3.    4.]
 [  5.    6.   nan   8.]
 [  0.   11.   12.  nan]]
---------------------------------------------
[[  1.    2.    3.    4. ]
 [  5.    6.    7.5   8. ]
 [  0.   11.   12.    6. ]]
```

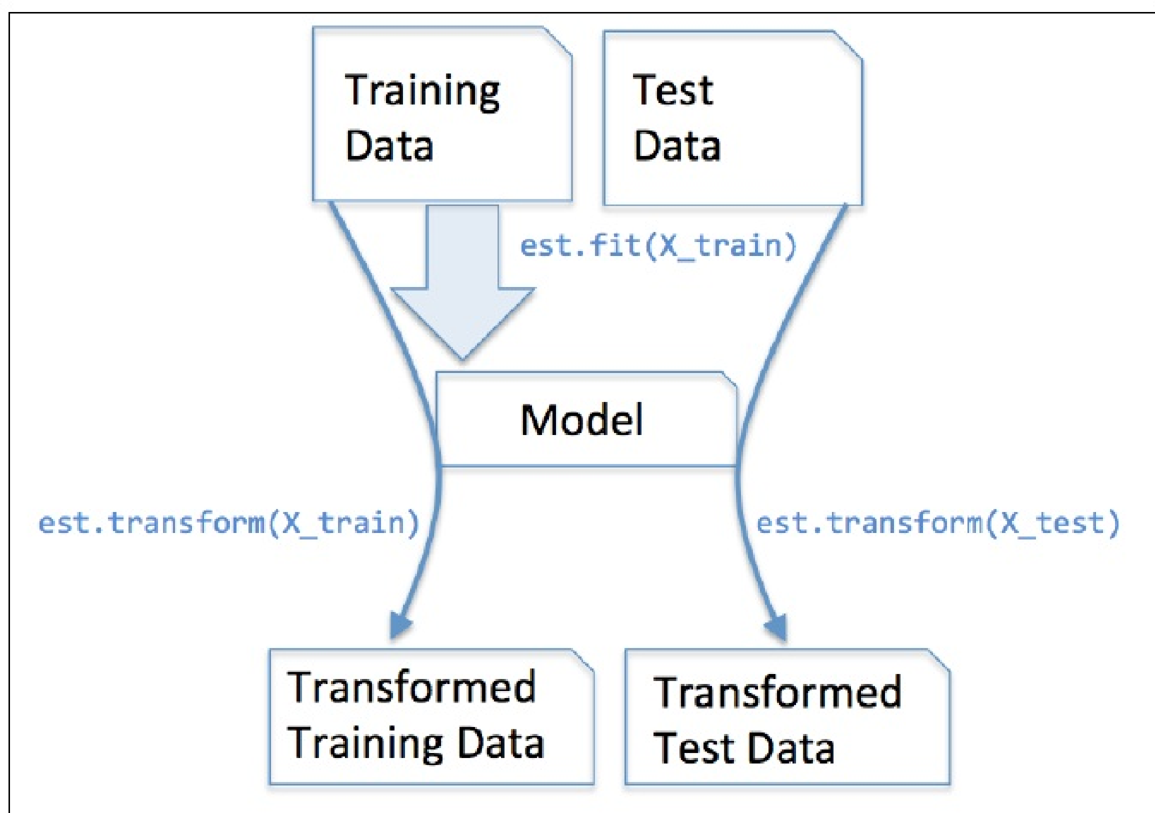
先计算每一列的平均值，然后用相应列的平均值来替换NaN。如果将参数axis=0改为axis=1，则会计算每个样本的所有特种的平均值。参数strategy的其他取值包括median和most_frequent。most_frequent对于处理分类数据类型的缺失值很有用。

理解sklearn中estimator的API

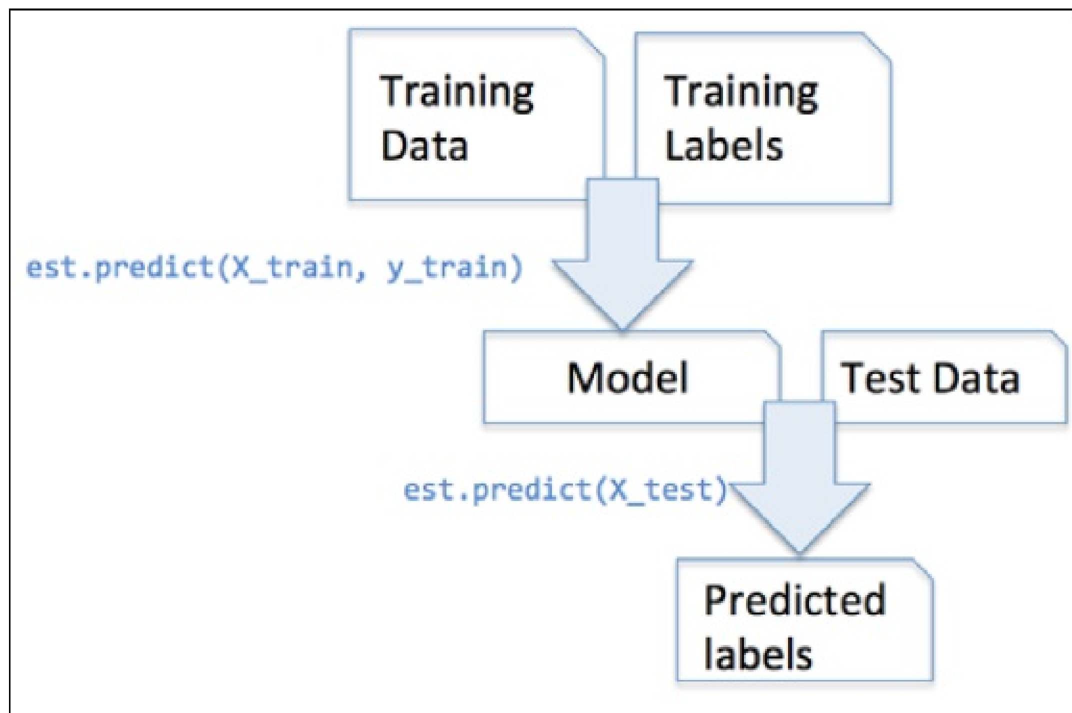
上一节我们调用了sklearn中Imputer类来处理缺失值。Imputer属于sklearn中所谓的transformer类，专门用于数据转换。此类estimator的两个必不可少的方法是fit和transform。fit方法用于从训练集中学习模型参数，transform用学习到的参数转换数据。

任何要进行转换的数据的特征维度必须和fit时的数据特征维度相同。

下图演示了fit和transform的过程：



我们在第三章用到的各类分类器属于sklearn中的estimator，它的API和transformer非常像。Estimator还有一个predict方法，大部分也含有transform方法。同样Estimator含有fit方法来学习模型参数。只不过不同的是，在监督学习时，我们还像fit方法提供每个样本的类别信息。



（图中应该是est.fit(X_train,y_train))

处理分类数据

目前为止，我们处理的都是数值型变量。但是真实世界的数据集通常都含有分类型变量(categorical value)的特征。当我们讨论分类型数据时，我们不区分其取值是否有序。比如T恤尺寸是有序的，因为XL>L>M。而T恤颜色是无序的。

在讲解处理分类数据的技巧之前，我们先创建一个新的DataFrame对象：

```
In  [20]: df = pd.DataFrame([['green', 'M', 10.1, 'class1'],
                            ['red', 'L', 13.5, 'class2'],
                            ['blue', 'XL', 15.3, 'class1']])
```

```
In  [21]: df.columns = ['color', 'size', 'price', 'classlabel']
```

```
In  [22]: df
```

Out[22]:

	color	size	price	classlabel
0	green	M	10.1	class1
1	red	L	13.5	class2
2	blue	XL	15.3	class1

上面创建的数据集含有无序特征(color)，有序特征(size)和数值型特征(price)。最后一列存储的是类别。在本书中类别信息都是无序的。

映射有序特征

为了保证学习算法能够正确解释有序特征(ordinal feature)，我们需要将分类型字符串转为整型数值。不幸地是，并没有能够直接调用的方法来自动得到正确顺序的size特征。因此，我们要自己定义映射函数。在接下来的简单的示例，假设我们知道特征取值间的不同，比如XL=L+1=M+2。

```
In [23]: size_mapping = {
             'XL':3,
             'L':2,
             'M':1
         }
```

```
In [24]: df['size'] = df['size'].map(size_mapping)
```

```
In [25]: df
```

Out[25]:

	color	size	price	classlabel
0	green	1	10.1	class1
1	red	2	13.5	class2
2	blue	3	15.3	class1

如果我们还想将整型变量转换回原来的字符串表示，我们还可以定义一个反映射字典 inv_size_mapping={v: k for k, v in size_mapping.items()}。

对类别进行编码

许多机器学习库要求类别是整型数值。虽然sklearn中大部分Estimator都能自动将类别转为整型，我还是建议大家手动将类别进行转换。对类别进行编码，和上一节中转化序列特征很相似。但不同的是类别是无序的，所以我们可以从0开始赋整数值:

```
In [29]: import numpy as np
```

```
In [30]: class_mapping = {label:idx for idx,label in enumerate(np.unique(df['classlabel']))}
```

```
In [31]: class_mapping
Out[31]: {'class1': 0, 'class2': 1}
```

接下来我们可以利用映射字典对类别进行转换：

```
In [32]: df['classlabel'] = df['classlabel'].map(class_mapping)
```

```
In [33]: df
```

Out[33]:

	color	size	price	classlabel
0	green	1	10.1	0
1	red	2	13.5	1
2	blue	3	15.3	0

得到整型类别值，也可以用映射字典转为原始的字符串值：

```
In  [34]: inv_class_mapping = {v: k for k, v in class_mapping.items()}
```

```
In  [35]: df['classlabel'] = df['classlabel'].map(inv_class_mapping)
```

```
In  [36]: df
```

Out[36]:

	color	size	price	classlabel
0	green	1	10.1	class1
1	red	2	13.5	class2
2	blue	3	15.3	class1

上面是我们自己手动创建的映射字典，sklearn中提供了LabelEncoder类来实现类别的转换：

```
In  [37]: from sklearn.preprocessing import LabelEncoder
```

```
In  [38]: class_le = LabelEncoder()
```

```
In  [39]: y = class_le.fit_transform(df['classlabel'].values)
```

```
In  [40]: y
Out[40]: array([0, 1, 0], dtype=int64)
```

fit_transform方法是fit和transform两个方法的合并。我们还可以调用inverse_transform方法得到原始的字符串类型值：

```
In  [41]: class_le.inverse_transform(y)
Out[41]: array(['class1', 'class2', 'class1'], dtype=object)
```

对离散特征进行独热编码

前面一节我们使用字典映射来转化有序特征，由于sklearn中Estimator把类型信息看做无序的，我们使用LabelEncoder来进行类别的转换。而对于无序的离散特征，我们也可以使用LabelEncoder来进行转换：

```
In  [42]: X = df[['color', 'size', 'price']].values

In  [43]: color_le = LabelEncoder()

In  [45]: X[:, 0] = color_le.fit_transform(X[:, 0])

In  [46]: X

Out[46]: array([[1L, 1L, 10.1],
                [2L, 2L, 13.5],
                [0L, 3L, 15.3]], dtype=object)
```

现在我们将无序离散特征转换为整型了，看起来下一步就是直接训练模型了。如果你这样想，恭喜你，你犯了一个很专业的错误。在处理分类型数据(categorical data)时，这是很常见的错误。你能发现问题所在吗？虽然“颜色”这一特征的值不含有顺序，但是由于进行了以下转换：

- blue → 0
- green → 1
- red → 2

学习算法会认为‘green’比‘blue’大，‘red’比‘green’大。而这显然是不正确的，因为本身颜色是无序的！模型错误的使用了颜色特征信息，最后得到的结果肯定不是我们想要的。

那么如何处理无序离散特征呢？常用的做法是独热编码**(one-hot encoding)**。独热编码会为每个离散值创建一个哑特征(dummy feature)。什么是哑特征呢？举例来说，对于‘颜色’这一特征中的‘蓝色’，我们将其编码为[蓝色=1，绿色=0，红色=0]，同理，对于‘绿色’，我们将其编码为[蓝色=0，绿色=1，红色=0]，特点就是向量只有一个1，其余均为0，故称之为one-hot。

在sklearn中，可以调用OneHotEncoder来实现独热编码：

```
In [47]: from sklearn.preprocessing import OneHotEncoder

In [48]: ohe = OneHotEncoder(categorical_features=[0])

In [49]: ohe.fit_transform(X).toarray()

Out[49]: array([[  0. ,   1. ,   0. ,   1. ,  10.1],
                [  0. ,   0. ,   1. ,   2. ,  13.5],
                [  1. ,   0. ,   0. ,   3. ,  15.3]])

In [ ]:

In [50]: X

Out[50]: array([[1L, 1L, 10.1],
                [2L, 2L, 13.5],
                [0L, 3L, 15.3]], dtype=object)

In [ ]:

In [51]: ohe.fit_transform(X)

Out[51]: <3x5 sparse matrix of type '<type 'numpy.float64'>'
             with 9 stored elements in COOrdinate format>
```

在初始化OneHotEncoder时，通过categorical_features参数设置要进行独热编码的列。还要注意的是OneHotEncoder的transform方法默认返回稀疏矩阵，所以我们调用toarray()方法将稀疏矩阵转为一般矩阵。我们还可以在初始化OneHotEncoder时通过参数sparse=False来设置返回一般矩阵。

除了使用sklearn中的OneHotEncoder类得到哑特征，我推荐大家使用pandas中的get_dummies方法来创建哑特征，get_dummies默认会对DataFrame中所有字符串类型的列进行独热编码：

```
In [52]: pd.get_dummies(df[['price', 'color', 'size']])
```

Out[52]:

	price	size	color_blue	color_green	color_red
0	10.1	1	0.0	1.0	0.0
1	13.5	2	0.0	0.0	1.0
2	15.3	3	1.0	0.0	0.0

将数据集分割为训练集和测试集

本节，我们使用一个新的数据集：Wine。Wine也属于UCI开源数据集，它包含178个样本，每个样本有13维度特征，描述了不同的化学属性。

Wine数据集

```
In [53]: df_wine = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-databases/wine/wine.data', header=None)

In [54]: df_wine.columns = ['Class label', 'Alcohol', 'Malic acid', 'Ash', 'Alcalinity of ash', 'Magnesium', 'Total phenols', 'Flavanoids',
                            'Nonflavanoid phenols', 'Proanthocyanins', 'Color intensity', 'Hue', 'OD280/OD315 of diluted wines', 'Proline']

In [55]: print('Class labels', np.unique(df_wine['Class label']))
         ('Class labels', array([1, 2, 3], dtype=int64))

In [56]: df_wine.head()
Out[56]:
```

	Class label	Alcohol	Malic acid	Ash	Alcalinity of ash	Magnesium	Total phenols	Flavanoids	Nonflavanoid phenols	Proanthocyanins	Color intensity	Hue	OD280/OD315 of diluted wines	Proli
0	1	14.23	1.71	2.43	15.6	127	2.80	3.06	0.28	2.29	5.64	1.04	3.92	1065
1	1	13.20	1.78	2.14	11.2	100	2.65	2.76	0.26	1.28	4.38	1.05	3.40	1050
2	1	13.16	2.36	2.67	18.6	101	2.80	3.24	0.30	2.81	5.68	1.03	3.17	1185
3	1	14.37	1.95	2.50	16.8	113	3.85	3.49	0.24	2.18	7.80	0.86	3.45	1480
4	1	13.24	2.59	2.87	21.0	118	2.80	2.69	0.39	1.82	4.32	1.04	2.93	735

Wine数据集一共有三个类别：1,2和3.表示三个葡萄品种。

首先将数据集随机分割为训练集和测试集，一种简单的方法是使用sklearn.cross_validation中的train_test_split方法：

```
In [57]: from sklearn.cross_validation import train_test_split

In [58]: X, y = df_wine.iloc[:, 1:].values, df_wine.iloc[:, 0].values

In [59]: X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=0)
```

解释一下上面的代码： 首先，我们将特征矩阵赋值给X，将类别向量赋值给y，然后调用train_test_split方法随机分割X和y。通过设置test_size=0.3,使得训练集占Wine样本数的70%，测试集占30%。

Note 在分割数据集时，如果确定训练集和测试集的大小没有通用的做法，一般我们选择60:40, 70:30或者80:20。对于大数据集，90:10甚至 99:1也是比较常见的。还要注意的是，通过本地验证得到最优模型和参数时，还要在整个数据集(训练集+验证集+测试集)上训练一次，得到最终的模型。

统一特征取值范围

特征缩放(feature scaling)是预处理阶段的关键步骤，但常常被遗忘。虽然存在决策树和随机森林这种是少数不需要特征缩放的机器学习算法，但对于大部分机器学习算法和优化算法来说，如果特征都在同一范围内，会获得更好的结果。比如第二章提到的梯度下降法。

特征缩放的重要性可以通过一个简单的示例解释。假设我们有两个特征，一个特征的取值范围是[1,10],另一个特征的取值范围是[1,100000]。我们使用Adaline中的平方误差函数，很明显，权重更新时会主要根据第二维度特征，这就使得在权重更新过程中第一个特征的话语权很小。另一个例子是如果kNN算法用欧氏距离作为距离度量，第二维度特征也占据了主要的话语权。

有两种方法能使不同的特征有相同的取值范围：归一化(normalization)和标准化(standardization)。两种方法还是有必要区分一下的。归一化指的是将特征范围缩放到[0,1]，是最小-最大缩放(min-max scaling)的特例。为了得到归一化结果，我们对每一个特征应用最小-最大缩放，计算公式如下：

$$x_{norm}^{(i)} = \frac{x^{(i)} - x_{\min}}{x_{\max} - x_{\min}}$$

其中，$x_{norm}^{(i)}$是$x^{(i)}$归一化后的结果，x_{min}是对应的列特征最小值，x_{max}则是最大值。

sklearn中实现了最小-最大缩放，调用MinMaxScaler类即可：

```
In [60]: from sklearn.preprocessing import MinMaxScaler

In [61]: mms = MinMaxScaler()

In [62]: X_train_norm = mms.fit_transform(X_train)

In [63]: X_test_norm = mms.transform(X_test)
```

虽然归一化方法简单，但相对来说，标准化对于大部分机器学习算法更实用。原因是大部分线性模型比如逻辑斯蒂回归和线性SVM在初始化权重参数时，要么选择0要么选择一个接近0的随机数。实用标准化，我们能将特征值缩放到以0为中心，标准差为1，换句话说，标准化后的特征形式服从正态分布，这样学习权重参数更容易。此外，标准化后的数据保持了异常值中的有用信息，使得算法对异常值不太敏感，这一点归一化就无法保证。

标准化的计算公式如下：

$$x_{std}^{(i)} = \frac{x^{(i)} - \mu_x}{\sigma_x}$$

此时，u_x是训练集对应特征列的平均值，是对应特征列的标准差。

下面一张表使用一个简单的例子，展示了标准化和归一化的区别：

input	standardized	normalized
0.0	-1.336306	0.0
1.0	-0.801784	0.2
2.0	-0.267261	0.4
3.0	0.267261	0.6
4.0	0.801784	0.8
5.0	1.336306	1.0

sklearn中提供了StandardScalar类实现列标准化：

```
In [64]: from sklearn.preprocessing import StandardScaler

In [65]: stdsc = StandardScaler()

In [66]: X_train_std = stdsc.fit_transform(X_train)

In [67]: X_test_std = stdsc.transform(X_test) #这里是transform()
```

再次强调，**StandardScaler**只使用训练集**fit**一次，这样保证训练集和测试集使用相同的标准进行的特征缩放。

选择有意义的特征

如果一个模型在训练集的表现比测试集好很多，那我们就要小心了，模型很可能过拟合了。过拟合意味着模型捕捉了训练集中的特例模式，但对未知数据的泛化能力比较差，我们也说模型此时具有高方差。

模型过拟合的一个原因是对于给定的训练集数据，模型过于复杂，常用的减小泛化误差的做法包括：

- 收集更多的训练集数据
- 正则化，即引入模型复杂度的惩罚项
- 选择一个简单点的模型，参数少一点的
- 降低数据的维度

在上面的一系列做法中，第一条收集更多数据通常不实用。在下一章，我们会学习一个有用的技巧来判断更多的训练集数据是否有帮助。在接下来的章节，我们学习正则化和特征选择的方法来降低过拟合。

L1正则

会议第三章，我们运用过L2正则来降低模型的复杂度，当时我们定义的L2正则项：

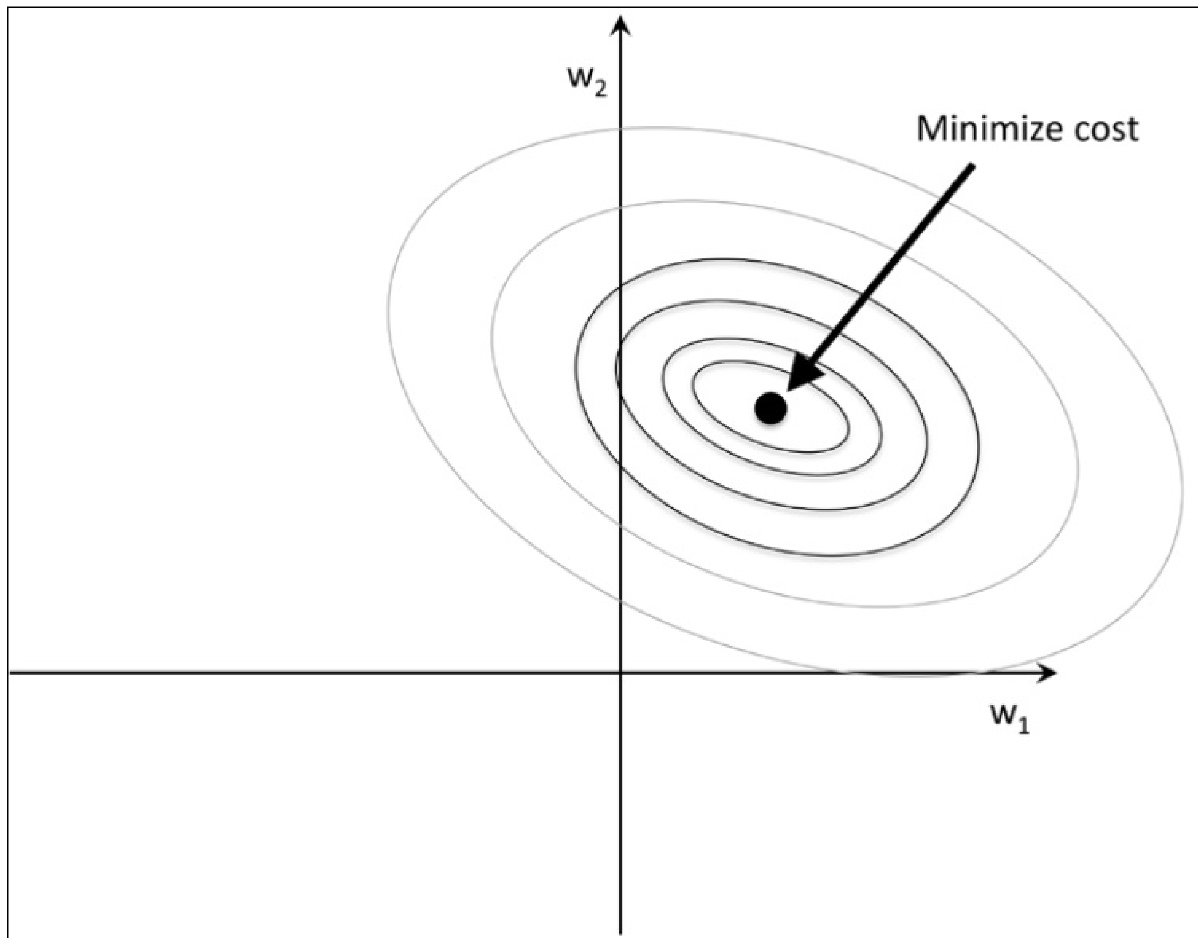
$$L2: \|w\|_2^2 = \sum_{j=1}^{m} w_j^2$$

除了L2正则，另一个中减低模型复杂度的方法是L1正则(L1 regularization)：

$$L1: \|w\|_1 = \sum_{j=1}^{m} |w_j|$$

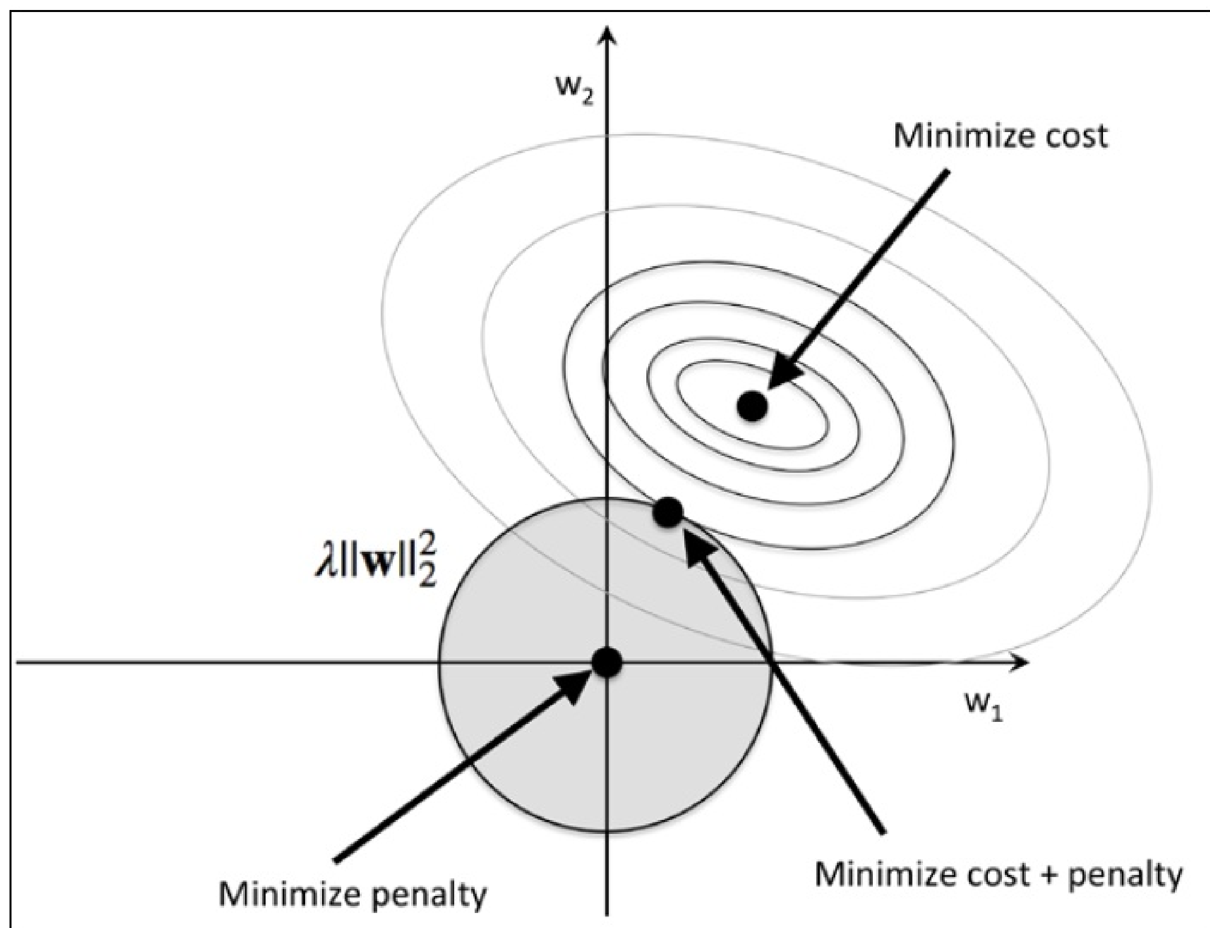
L2正则项是权重参数的平方和，而L1正则项是权重参数的绝对值和。相对于L2, L1正则项趋向于得到稀疏特征向量，即很多特征权重参数为0.如果数据集的特征维度很高且特征不相干(极端情况是 不相干的特征维度数目比训练样本数还大)，特征稀疏性是非常有用的。 由于很多特征权重为0，所以，很多人也把L1正则看做特征选择的一种方式。

为了更好地理解L1正则倾向于产生稀疏特征，让我们看一下正则化的几何解释。假设损失函数是差平方损失函数(sum of the squared errors, SSE),且只含有两个权重参数w_1, w_2，易知损失函数是凸函数。对于给定的损失函数，我们的目的是找到损失函数取最小值时对应的权重值，如下图损失函数等高线所示，当(w_1, w_2)取椭圆中心点时，损失函数值最小：



而正则项是对现在损失函数的惩罚项，它鼓励权重参数取小一点的值，换句话说，正则项惩罚的是大权重参数。

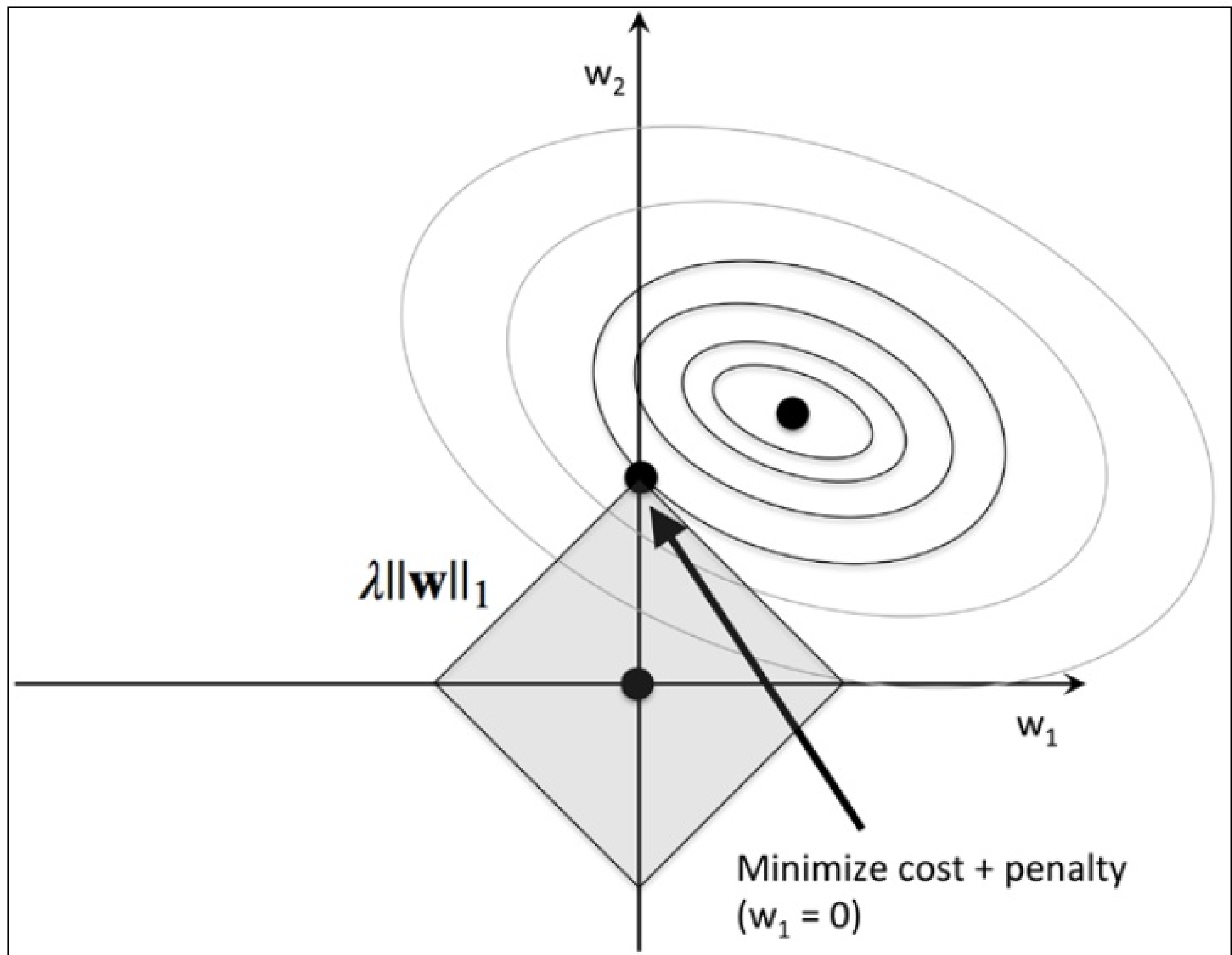
因此，如果增大正则系数 的值，也就增大了正则项的威力，也就会导致权重参数变小(趋向于0)，从而也减小了模型对训练数据的依赖。我们在下图画出L2惩罚项：



L2正则项用图上阴影球形表示 。正则化后的 新损失函数=原始损失函数+正则项，我们可以换一种思路解释这个公式，新的损失函数还是原始损失函数，我们把正则项看做权重参数的限制条件，也就是说，权重参数的取值范围必须在图中阴影球形内。如果增大 的值，会缩小阴影球形面积。比如，如果 趋向正无穷，则阴影面积趋向0，权重参数也趋向0.

我们总结一下：我们的目标是最小化原始损失函数和正则项，等价于在原始损失函数基础上增加限制条件，更加偏向于简单模型来减小方差。

现在我们讨论L1正则项和稀疏性。L1正则和刚才讨论的L2正则很像。当然区别还是有的：L1正则是权重参数绝对值的和，我们用一个另行区域表示，如下图所示：



由于菱形的特点，在和损失函数等高线相交时，最小点很可能落在坐标轴上，这就导致了特征的稀疏性。如果你对这背后的数学感兴趣，推荐阅读 *The Elements of Statistical Learning, Trevor Hastie, Robert Tibshirani, and Friedman, Springer*。

对于sklearn中那些支持L1正则的模型，我们只需要在初始化时用penalty参数设置为L1正则即可：

```
In  [63]: from sklearn.linear_model import LogisticRegression

In  [64]: LogisticRegression(penalty='l1')
Out[64]: LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
                   penalty='l1', random_state=None, solver='liblinear', tol=0.0001,
                   verbose=0, warm_start=False)
```

将L1正则逻辑斯蒂回归应用到标准化后的Wine数据集：

```
In [65]: lr = LogisticRegression(penalty='l1', C=0.1)

In [66]: lr.fit(X_train_std, y_train)

Out[66]: LogisticRegression(C=0.1, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
                   penalty='l1', random_state=None, solver='liblinear', tol=0.0001,
                   verbose=0, warm_start=False)

In [67]: print('Training accuracy:', lr.score(X_train_std, y_train))

         ('Training accuracy:', 0.9838709677419355)

In [69]: print('Test accuracy:', lr.score(X_test_std, y_test))

         ('Test accuracy:', 0.9814814814814815)
```

模型在训练集和测试集上的准确率说明没有过拟合。如果我们调用lr.intercept_属性，可以发现只返回了三个值：

```
In [70]: lr.intercept_

Out[70]: array([-0.38380993, -0.15808033, -0.70046381])
```

由于Wine数据集是多类别数据，所以lr使用了One-vs-Rest(OvR)方法，所以上面三个值分别属于三个模型：第一个模型用类别1 vs 类别2和3；第二个模型用类别2 vs 类别1和3；第三个模型用类别3 vs 类别1和2。

```
In [71]: lr.coef_

Out[71]: array([[ 0.28004979,  0.        ,  0.        , -0.02788486,  0.        ,
                  0.        ,  0.71000739,  0.        ,  0.        ,  0.        ,
                  0.        ,  0.        ,  1.23666102],
                [-0.643982  , -0.06880393, -0.05720686,  0.        ,  0.        ,
                  0.        ,  0.        ,  0.        ,  0.        , -0.92675283,
                  0.06019256,  0.        , -0.37104661],
                [ 0.        ,  0.06155493,  0.        ,  0.        ,  0.        ,
                  0.        , -0.63530254,  0.        ,  0.        ,  0.49779685,
                 -0.35825682, -0.57213556,  0.        ]])
```

通过lr.coef_得到权重数组，共三行，每一个类对应一行。每一行有13个参数，对应13个特征。网络输入计算如下：

$$z = w_1 x_1 + \cdots + w_m x_m = \sum_{j=0}^{m} x_j w_j = \boldsymbol{w}^T \boldsymbol{x}$$

我们可以发现权重向量中有很多0值，这说明L1正则可以作为特征选择的一种手段，得到的模型具有鲁棒性。

最后，我们画出正则路径，即不同正则威力下的不同特征的权重参数：

```
In [72]: import matplotlib.pyplot as plt
```

```
In [73]: fig = plt.figure()
```

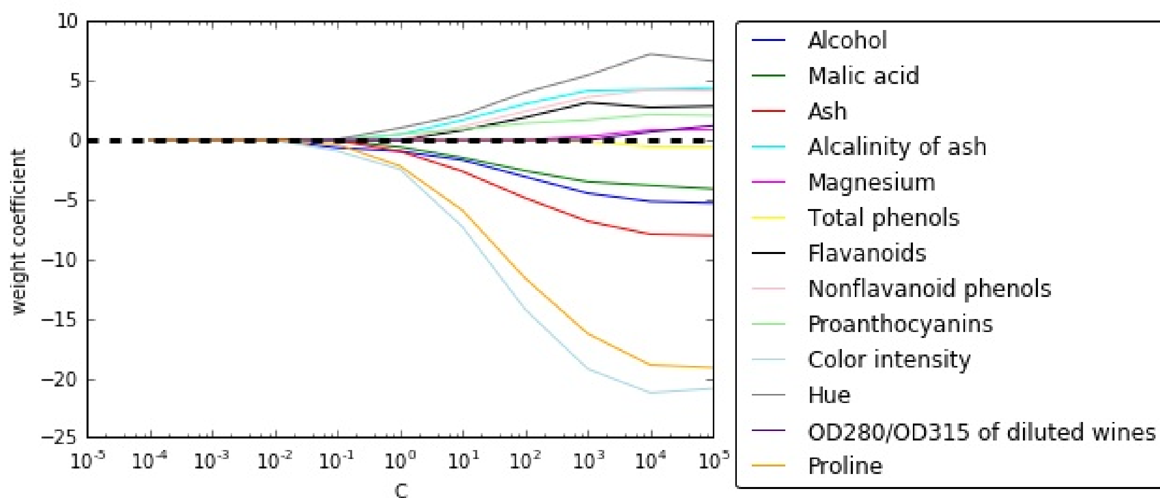
```
<matplotlib.figure.Figure at 0x1da16240>
```

```
In [78]: ax = plt.subplot(111)
colors = ['blue', 'green', 'red', 'cyan', 'magenta', 'yellow', 'black', 'pink', 'lightgreen', 'lightblue',
          'gray', 'indigo', 'orange']
weights, params = [], []

for c in np.arange(-4, 6):
    lr = LogisticRegression(penalty='l1', C=10**c, random_state=0)
    lr.fit(X_train_std, y_train)
    weights.append(lr.coef_[1])
    params.append(10 ** c)

weights = np.array(weights)
for column, color in zip(range(weights.shape[1]), colors):
    plt.plot(params, weights[:, column], label=df_wine.columns[column+1], color=color)


plt.axhline(0, color='black', linestyle='--', linewidth=3)
plt.xlim([10**(-5), 10**5])
plt.ylabel('weight coefficient')
plt.xlabel('C')
plt.xscale('log')
plt.legend(loc='upper left')
ax.legend(loc='upper center', bbox_to_anchor=(1.38, 1.03), ncol=1, fancybox=True)
plt.show()
```



我们可以发现，如果C<0.1,正则项威力很大时，所有特征权重都为0，□。

序列特征选择算法

另一种减小模型复杂度和避免过拟合的方法是通过特征选择进行维度降低(dimensionality reduction)，这个方法尤其对非正则模型有用。维度降低有两种做法：特征选择(feature selection)和特征抽取(feature extraction)。

特征选择会从原始特征集中选择一个子集合。特征抽取是从原始特征空间抽取信息，从而构建一个新的特征子空间。本节，我们学习特征选择算法。在下一章，我们会学到不同的特征抽取方法来将数据集压缩到一个低维度特征子空间。

序列特征选择算法属于贪心搜索算法，用于将原始的d维度特征空间降低到k维度特征子空间，其中k<d。

特征选择算法的原理是自动选择一个特征子集，子集中的特征都是和问题最相关的特征，这样能够提高计算效率并且由于溢出了不相干特征和噪音也降低了模型的泛化误差。

一个经典的序列特征选择算法是序列后向选择(sequential backward selection, SBS),它能够降低原始特征维度提高计算效率，在某些情况下，如果模型过拟合，使用SBS后甚至能提高模型的预测能力。

Note贪心算法在每一次选择时都会做出局部最优选择，通常产生一个次优全局解，暴力搜索要考虑所有的可能的情况所以会保证得到全局最优解。但由于穷搜的计算复杂度过高，导致其并不是最佳选择。

SBS算法的idea很简单：SBS序列地从原始特征集中移除特征，直到新的特征集数目达到事先确定的值。而在移除特征时，我们需要定义评价函数J。一个特征的重要性就用特征移除后的评价函数值表示。我们每一次都把那些评价函数值最大的特征移除，也就是那些对评价函数影响最小的特征去掉。所以，SBS算法有以下4个步骤：

- 1 初始化k=d，其中d是原始特征维度。
- 2 确定那个评价函数最大的特征 。
- 3 从X_k中移除特征x^-, k=k-1。
- 4 如果k等于事先确定的阈值则终止；否则回到步骤2。

不过，sklearn目前并没有实现SBS算法，好在算法简单，我们可以自己实现：

```
In [79]: from sklearn.base import clone
         from itertools import combinations
         import numpy as np
         from sklearn.cross_validation import train_test_split
         from sklearn.metrics import accuracy_score
```

```
In [80]: class SBS():
             def __init__(self, estimator, k_features, scoring=accuracy_score, test_size=0.25, random_state=1):
                 self.scoring = scoring
                 self.estimator = clone(estimator)
                 self.k_features = k_features
                 self.test_size = test_size
                 self.random_state = random_state

             def fit(self, X, y):
                 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=self.test_size, random_state=self.random_state)
                 dim = X_train.shape[1]
                 self.indices_ = tuple(range(dim))
                 self.subsets_ = [self.indices_]
                 score = self._calc_score(X_train, y_train, X_test, y_test, self.indices_)
                 self.scores_ = [score]

                 while dim > self.k_features:
                     scores = []
                     subsets = []

                     for p in combinations(self.indices_, r=dim-1):
                         score = self._calc_score(X_train, y_train, X_test, y_test, p)
                         scores.append(score)
                         subsets.append(p)

                     best = np.argmax(scores)
                     self.indices_ = subsets[best]
                     self.subsets_.append(self.indices_)
                     dim -= 1
                     self.scores_.append(scores[best])
                 self.k_score_ = self.scores_[-1]

                 return self
         def transform(self, X):
             return X[:, self.indices_]

         def _calc_score(self, X_train, y_train,  X_test, y_test, indices):
             self.estimator.fit(X_train[:, indices], y_train)
             y_pred = self.estimator.predict(X_test[:, indices])
             score = self.scoring(y_test, y_pred)
             return score
```

在上面的代码中，我们定义了k_features参数来设定想要得到的特征子集数。使用accuracy_score来评估模型在特征子集的表现。在fit方法的while循环内，通过itertools.combination创建特征子集然后对其评估，

现在我们用KNN作为Estimator来运行SBS算法：

```
In [100]: from sklearn.neighbors import KNeighborsClassifier

In [101]: import matplotlib.pyplot as plt

In [102]: knn = KNeighborsClassifier(n_neighbors=2)

In [103]: sbs = SBS(knn, k_features=1)

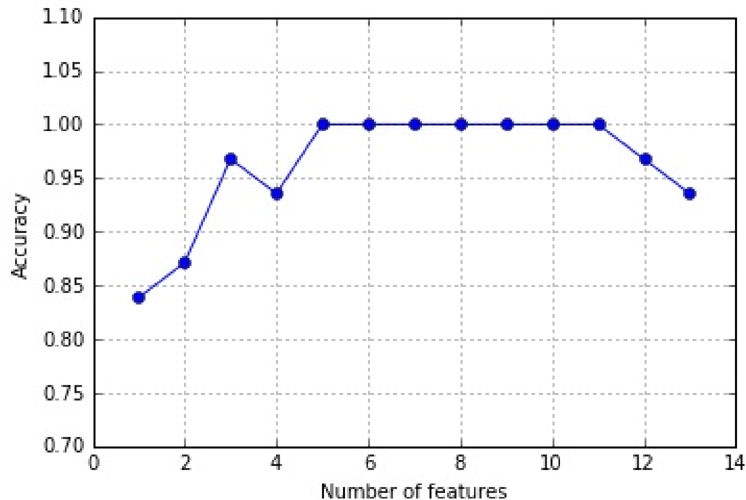
In [104]: sbs.fit(X_train_std, y_train)
Out[104]: <__main__.SBS instance at 0x000000001D272748>
```

虽然SBS的fit方法中有分割数据集的功能，但我们还是为SBS提供了训练集，然后fit方法将其分割为子训练集和子测试集，而这个子测试集被称为验证集(validation dataset)。

SBS算法记录了每一步最优特征子集的成绩，我们画出每个最优特征子集在验证集上的分类准确率：

In [105]:
```
k_feat = [len(k) for k in sbs.subsets_]
```

In [106]:
```
plt.plot(k_feat, sbs.scores_, marker='o')
plt.ylim([0.7, 1.1])
plt.ylabel('Accuracy')
plt.xlabel('Number of features')
plt.grid()
plt.show()
```



我们可以看到，最开始随着特征数目的减少，分类准确率一直在提高，原因可能是降低了维度诅咒。对于k={5,6,7,8,9,10,11}，分类准确率是100%.

我们将最优的5维度特征打印出来看一下：

In [107]:
```
k5 = list(sbs.subsets_[8])
```

In [108]:
```
print(df_wine.columns[1:][k5])
```

```
Index([u'Alcohol', u'Malic acid', u'Alcalinity of ash', u'Hue', u'Proline'], dtype='object')
```

接下来我们使用整个特征维度来检验KNN模型在测试集的分类准确率：

In [109]:
```
knn.fit(X_train_std, y_train)
```

Out[109]:
```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
           metric_params=None, n_jobs=1, n_neighbors=2, p=2,
           weights='uniform')
```

In [110]:
```
print('Training accuracy:', knn.score(X_train_std, y_train))
```

```
('Training accuracy:', 0.9838709677419355)
```

In [111]:
```
print('Test accuracy:', knn.score(X_test_std, y_test))
```

```
('Test accuracy:', 0.9444444444444442)
```

、、

可以看到使用原始的特征集建模，在训练集上分类准确率98%，测试集上分类准确率94%，可能是模型有一丢丢过拟合。

我们再使用最优的5维度特征建模看看：

```
In [112]: knn.fit(X_train_std[:, k5], y_train)

Out[112]: KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                     metric_params=None, n_jobs=1, n_neighbors=2, p=2,
                     weights='uniform')
```

```
In [114]: print('Training accuracy:', knn.score(X_train_std[:, k5], y_train))

          ('Training accuracy:', 0.95967741935483875)
```

```
In [115]: print('Test accuracy:', knn.score(X_test_std[:, k5], y_test))

          ('Test accuracy:', 0.96296296296296291)
```

使用不到一半的原始特征，虽然在训练集上分类准确率下降了，但是在测试集上的表现却提高了！并且此时训练集和测试集准确率相差不多，我们很好地降低了过拟合。

Note sklearn中提供了很多特征选择算法。包括基于特征参数的递归后向消除法，基于树方法提供的特征重要性的特征选择法和单变量统计检验。具体的可以看sklearn文档。

利用随机森林评估特征重要性

在前面一节，你学习了如何利用L1正则将不相干特征变为0，使用SBS算法进行特征选择。另一种从数据集中选择相关特征的方法是利用随机森林。

随机森林能够度量每个特征的重要性，我们可以依据这个重要性指标进而选择最重要的特征。sklearn中已经实现了用随机森林评估特征重要性，在训练好随机森林模型后，直接调用feature_importances属性就能得到每个特征的重要性。

下面用Wine数据集为例，我们训练一个包含10000棵决策树的随机森林来评估13个维度特征的重要性(第三章我们就说过，对于基于树的模型，不必对特征进行标准化或归一化):

```
In [81]: from sklearn.ensemble import RandomForestClassifier

In [82]: feat_labels = df_wine.columns[1:]

In [83]: forest = RandomForestClassifier(n_estimators=10000, random_state=0, n_jobs=-1)

In [84]: forest.fit(X_train, y_train)

Out[84]: RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
                     max_depth=None, max_features='auto', max_leaf_nodes=None,
                     min_samples_leaf=1, min_samples_split=2,
                     min_weight_fraction_leaf=0.0, n_estimators=10000, n_jobs=-1,
                     oob_score=False, random_state=0, verbose=0, warm_start=False)

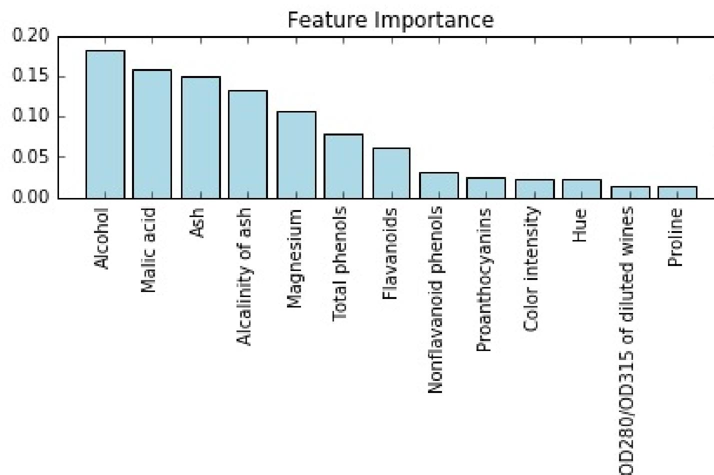
In [85]: importances = forest.feature_importances_

In [86]: indices = np.argsort(importances)[::-1]

In [89]: for f in range(X_train.shape[1]):
             print("%2d) %-*s %f" % (f + 1, 30, feat_labels[f], importances[indices[f]]))

          1) Alcohol                        0.182483
          2) Malic acid                     0.158610
          3) Ash                            0.150948
          4) Alcalinity of ash              0.131987
          5) Magnesium                      0.106589
          6) Total phenols                  0.078243
          7) Flavanoids                     0.060718
          8) Nonflavanoid phenols           0.032033
          9) Proanthocyanins                0.025400
         10) Color intensity                0.022351
         11) Hue                            0.022078
         12) OD280/OD315 of diluted wines   0.014645
         13) Proline                        0.013916
```

In [90]:
```
plt.title('Feature Importance')
plt.bar(range(X_train.shape[1]),
        importances[indices],
        color='lightblue',
        align='center')
plt.xticks(range(X_train.shape[1]),
           feat_labels, rotation=90)
plt.xlim([-1, X_train.shape[1]])
plt.tight_layout()
plt.show()
```



我们可以得出结论：'Alcohol'是最能区分类别的特征。有趣地是，重要性排名前三的特征也在SBS的最优5特征子集中。

sklearn的随机森林实现，包括一个transform方法能够基于用户给定的阈值进行特征选择，所以如果你要用RandomFroestClassifier作为特征选择器，这就很easy了。举个例子：设置阈值为0.15，会选择出三个维度特征，Alcohol、Malic acid和Ash。

In [84]:
```
X_selected = forest.transform(X_train, threshold=0.15)
```

```
c:\python27\lib\site-packages\sklearn\utils\__init__.py:93: DeprecationWarning: Function transform
 as feature selectors will be removed in version 0.19. Use SelectFromModel instead.
  warnings.warn(msg, category=DeprecationWarning)
```

In [85]:
```
X_selected.shape
```

Out[85]:
```
(124L, 3L)
```

总结

本章最开始我们介绍了如何处理缺失值。在训练模型之前，我们必须保证已经正确处理分类数据。

此外，我们简单讨论了L1正则，它可以帮助我们降低模型复杂度来避免过拟合。另一种移除不相关特征的方法是使用序列特征选择算法来选择有意义的特征。

在下一章，你将会学到另一类降维的方法：特征抽取。它能够将特征压缩到一个低维度子空间而不是移除某些特征。

第五章 通过降维压缩数据

在第四章，你学习了使用不同的特征选择方法来降维，除了特征选择，另一种降维的方法是特征抽取(feature extraction)。本章你将会学到三种基本的方法，帮助我们摘要数据集的信息，即将原始的特征空间压缩到低纬度的特征子空间。数据压缩是机器学习中的重要课题，它能帮助我们存储和分析当今时代不断涌现的大数据。本章，我们主要讨论以下几个内容：

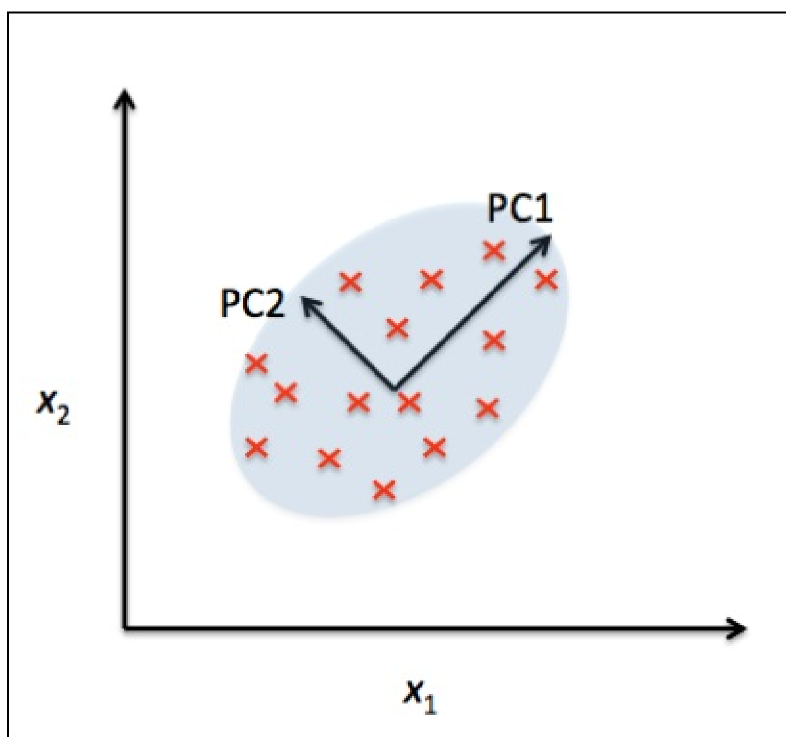
- 主成分分析（principal component analysis, PCA), 用于无监督数据压缩
- 线性判别分析(linear discriminant analysis, LDA), 用于监督降维作为一种监督降维
- 通过核PCA进行非线性降维

PCA进行无监督降维

类似于特征选择，我们可以使用特征抽取来减小数据集中的特征维度。不过，不同于特征选择算法会保留原始特征空间，特征抽取会将原始特征转换/映射到一个新的特征空间。换句话说，特征抽取可以理解为是一种数据压缩的手段，同时保留大部分相关信息(译者注：理解为摘要)。特征抽取是用于提高计算效率的典型手段，另一个好处是也能够减小维度诅咒(curse of dimensionality)，特别是对于没有正则化的模型。

PCA(principal component analysis, 主成分分析)是一种被广泛使用的无监督的线性转换技术，主要用于降维。其他领域的应用还包括探索数据分析和股票交易的信号去噪，基因数据分析和基因表达。

PCA根据特征之间的相关性帮助我们确定数据中存在的模式。简而言之，PCA的目标是找到高维数据中最大方差的方向，并且将高维数据映射到一个新的子空间，这个子空间的方向不大于原始特征空间。新子空间的正交轴(主成分)可以被解释为原始空间的最大方差方向。下图中x_1, x_2是原始特征轴，$PC1 PC2$是主成分：



如果使用PCA降维，我们需要构造一个d*k维的转换矩阵W，它能将样本向量x映射到新的k维度的特征子空间，k<<d:

$$\boldsymbol{x} = [x_1, x_2, \ldots, x_d], \quad \boldsymbol{x} \in \mathbb{R}^d$$

$$\downarrow \boldsymbol{xW}, \quad \boldsymbol{W} \in \mathbb{R}^{d \times k}$$

$$\mathbf{z} = [z_1, z_2, \ldots, z_k], \quad \mathbf{z} \in \mathbb{R}^k$$

映射后的子空间，第一主成分包含最大的方差，第二主成分包含次大的方差，以此类推，并且各个主成分方向是正交的(不相关)。

PCA方向极其容易受到数据中特征范围影响，所以在运用**PCA**前一定要做特征标准化，这样才能保证每维度特征的重要性等同。

再详细讲解PCA的细节之前，我们先介绍 PCA的步骤：

- 1 将d维度原始数据标准化。
- 2 构建协方差矩阵。
- 3 求解协方差矩阵的特征向量和特征值。
- 4 选择值最大的k个特征值对应的特征向量，k就是新特征空间的维度，k<<d。
- 5 利用k特征向量构建映射矩阵W。
- 6 将原始d维度的数据集X，通过映射矩阵W转换到k维度的特征子空间。

聊一聊方差

本节，我们会学习PCA中的前四个步骤：标准化数据、构建协方差矩阵、得到特征值和特征向量以及对特征值排序得到排名靠前的特征向量。

数据集还是用第四章介绍过的Wine数据集，先将原始Wine分割为训练集和测试集，然后标准化：

```
In [2]: %matplotlib inline
        import numpy as np
        import pandas as pd
        import matplotlib.pyplot as plt

In [4]: df_wine = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-databases/wine/wine.data', header=None)

In [ ]:

In [5]: from sklearn.cross_validation import train_test_split
        from sklearn.preprocessing import StandardScaler

In [6]: X, y = df_wine.iloc[:, 1:].values, df_wine.iloc[:, 0].values

In [7]: X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=0)

In [8]: sc = StandardScaler()

In [9]: X_train_std = sc.fit_transform(X_train)

In [10]: X_test_std = sc.transform(X_test)
```

上面的代码完成了PCA的第一步，对数据进行标准化。我们再来看第二步：构建协方差矩阵。协方差矩阵是对称矩阵，d*d维度，其中d是原始数据的特征维度，协方差矩阵的每个元素是两两特征之间的协方差。

举个例子，特征$x_j x_k$的协方差计算公式如下：

$$\sigma_{jk} = \frac{1}{n}\sum_{i=1}^{n}\left(x_j^{(i)} - \mu_j\right)\left(x_k^{(i)} - \mu_k\right)$$

其中，$u_j u_k$分别是样本中第j维度特征和第k维度特征的平均值。由于我们已经将数据标准化了，所以这里$u_j = u_k = 0$。如果☐意味着j和k这两维度特征值要么同时增加要么同时衰减，反之☐,意味着这两个特征的值变化方向相反。

假设数据有三维度特征，协方差矩阵如下：

$$\sum=\begin{bmatrix}\sigma_1^2 & \sigma_{12} & \sigma_{13}\\ \sigma_{21} & \sigma_2^2 & \sigma_{23}\\ \sigma_{31} & \sigma_{32} & \sigma_3^2\end{bmatrix}$$

协方差矩阵的特征向量代表了主成分(最大方差的方向)，对应的特征值决定了特征向量绝对值的大小。在Wine数据集对应的13*13的协方差矩阵，我们会得到13个特征值。

如何得到特征值和特征向量呢？会议线性代数课上讲过的内容：

$$\Sigma v = \lambda v$$

其中，。我们当然不可能手算特征值，NumPy提供了linalg.eig函数用于得到特征值和特征向量：

```
In [11]: import numpy as np
```

```
In [12]: cov_mat = np.cov(X_train_std.T)
```

```
In [13]: eigen_vals, eigen_vecs = np.linalg.eig(cov_mat)
```

```
In [14]: print('\nEigenvalues \n%s' %eigen_vals)
```

```
Eigenvalues
[ 4.8923083   2.46635032  1.42809973  1.01233462  0.84906459  0.60181514
  0.52251546  0.08414846  0.33051429  0.29595018  0.16831254  0.21432212
  0.2399553 ]
```

我们先使用np.cov方法得到数据的协方差矩阵，然后利用linalg.eig方法计算出特征向量(eigen_vecs)和特征值(eigen_vals)。

由于我们的目的是数据压缩，即降维，所以我们只将那些包含最多信息(方差)的特征向量(主成分)拿出来。什么样的特征向量包含的信息最多呢？这就要看特征值了，因为特征值定义了特征向量的大小，我们先对特征值进行降序排序，前k个特征值对应的特征向量就是我们要找的主成分。

我们再学一个概念：方差解释率(variance explained ration)。一个特征值的方差解释率就是次特征值在特征值总和的占比：

$$\frac{\lambda_j}{\sum_{j=1}^{d} \lambda_j}$$

利用NumPy提供的cumsum函数，我们可以计算累计解释方差和：

```
In [15]: tot = sum(eigen_vals)
```

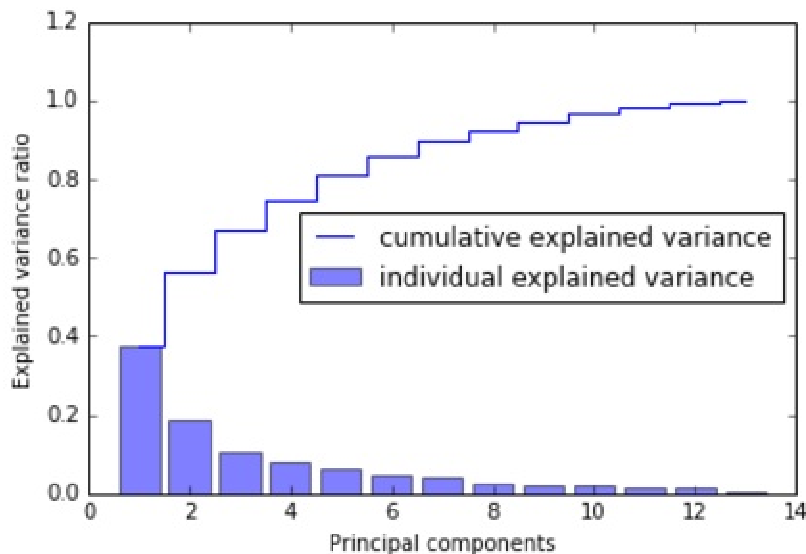
```
In [16]: var_exp = [(i / tot) for i in sorted(eigen_vals, reverse=True)]
```

```
In [17]: cum_var_exp = np.cumsum(var_exp)
```

```
In [ ]:
```

```
In [18]: import matplotlib.pyplot as plt
```

```
In [20]: plt.bar(range(1, 14), var_exp, alpha=0.5, align='center',
                 label='individual explained variance')
         plt.step(range(1, 14), cum_var_exp, where='mid',
                  label='cumulative explained variance')
         plt.ylabel('Explained variance ratio')
         plt.xlabel('Principal components')
         plt.legend(loc='best')
         plt.show()
```



从上面的结果图我们可以看到第一个主成分占了近40%的方差(信息)，前两个主成分占了60%的方差。

很多同学看到这里，可能将方差解释率和第四章讲到的用随机森林评估特征重要性联系起来，二者还是有很大区别的，PCA是一种无监督方法，在整个计算过程中我们都没有用到类别信息！随机森林是监督模型，建模时用到了类别信息。

方差的物理含义是对值沿着特征轴的传播进行度量。

特征转换

在得到特征向量后，接下来我们就可以对原始特征进行转换了。本节我们先对特征值进行降序排序，然后用特征向量构建映射矩阵，最后用映射矩阵将原始数据映射到低维度特征子空间。

先对特征值排序：

```
In [23]: eigen_pairs = [(np.abs(eigen_vals[i]), eigen_vecs[:, i]) for i in range(len(eigen_vals))]
```

```
In [25]: eigen_pairs.sort(reverse=True)
```

接下来，我们选择最大的两个特征值对应的特征向量，这里只用两个特征向量是为了下面画图方便，在实际运用PCA时，到底选择几个特征向量，要考虑到计算效率和分类器的表现两个方面(译者注：常用的选择方式是特征值子集要包含90%方差)：

```
In [26]: w = np.hstack((eigen_pairs[0][1][:, np.newaxis],
                        eigen_pairs[1][1][:, np.newaxis]))
```

```
In [27]: print('Matrix W: \n', w)

('Matrix W: \n', array([[ 0.14669811,  0.50417079],
       [-0.24224554,  0.24216889],
       [-0.02993442,  0.28698484],
       [-0.25519002, -0.06468718],
       [ 0.12079772,  0.22995385],
       [ 0.38934455,  0.09363991],
       [ 0.42326486,  0.01088622],
       [-0.30634956,  0.01870216],
       [ 0.30572219,  0.03040352],
       [-0.09869191,  0.54527081],
       [ 0.30032535, -0.27924322],
       [ 0.36821154, -0.174365  ],
       [ 0.29259713,  0.36315461]]))
```

现在，我们创建了一个132的的映射矩阵*W*。然后对样本(*1*13维度)进行映射，就能得到2维度的新样本：

$$x' = xW$$

```
In [28]: X_train_std[0].dot(w)

Out[28]: array([ 2.59891628,  0.00484089])
```

直接对原始数据(124*13)映射：

$$X' = XW$$

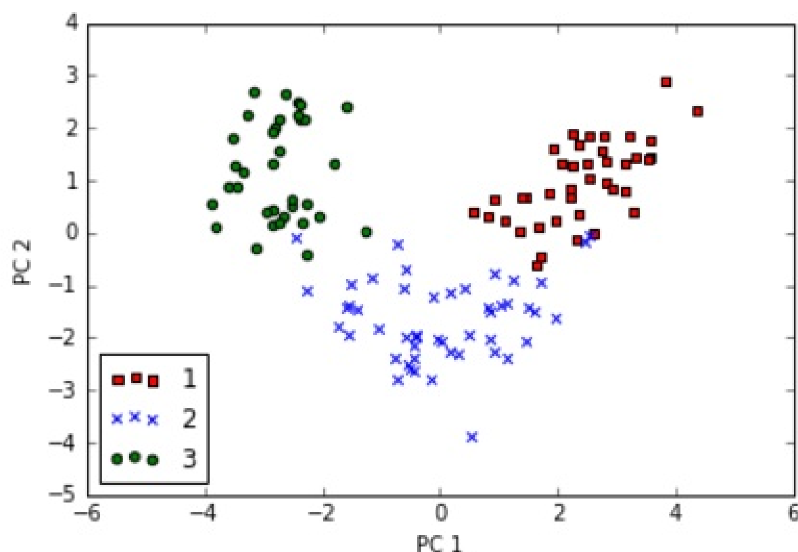
```
In [23]: X_train_pca = X_train_std.dot(w)
```

特征维度降到2维度后，我们就可以用散点图将数据可视化出来了：

```
In [35]: colors = ['r', 'b', 'g']
```

```
In [36]: markers = ['s', 'x', 'o']
```

```
In [38]: for l, c, m in zip(np.unique(y_train), colors, markers):
             plt.scatter(X_train_pca[y_train==l, 0],
                         X_train_pca[y_train==l, 1],
                         c=c, label=l, marker=m)
         plt.xlabel('PC 1')
         plt.ylabel('PC 2')
         plt.legend(loc='lower left')
         plt.show()
```



从上图可以看到，数据在x轴(第一主成分)上要比y轴(第二主成分)分布更广，这也符合方差解释率的结果。数据降维后，直觉上使用线性分类器就能够将数据分类。

scikit-learn中的PCA

上一小节我们详细讨论了PCA的步骤，在实际应用时，一般不会使用自己实现，而是直接调用sklearn中的PCA类，PCA类是另一个transformer类：我们先用训练集训练模型参数，然后统一应用于训练集和测试集。

下面我们就是用sklearn中的PCA类对Wine数据降维，然后调用逻辑斯蒂回归模型分类，最后将决策界可视化出来：

In [39]:
```
from matplotlib.colors import ListedColormap
```

In [41]:
```
def plot_decision_regions(X, y, classifier, resolution=0.02):
    #setup marker generator and color map
    markers = ('s', 'x', 'o', '^', 'v')
    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
    cmap = ListedColormap(colors[:len(np.unique(y))])

    #plot the decision surface
    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                           np.arange(x2_min, x2_max, resolution))
    Z = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    Z = Z.reshape(xx1.shape)

    plt.contourf(xx1, xx2, Z, alpha=0.4, cmap=cmap)
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())

    #plot class samples
    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(x=X[y==cl, 0], y=X[y==cl, 1],
                    alpha=0.8, c=cmap(idx),
                    marker=markers[idx], label=cl)
```

In [42]:
```
from sklearn.linear_model import LogisticRegression
```

In [43]:
```
from sklearn.decomposition import PCA
```

In [44]:
```
pca = PCA(n_components=2)
```

In [45]:
```
lr = LogisticRegression()
```

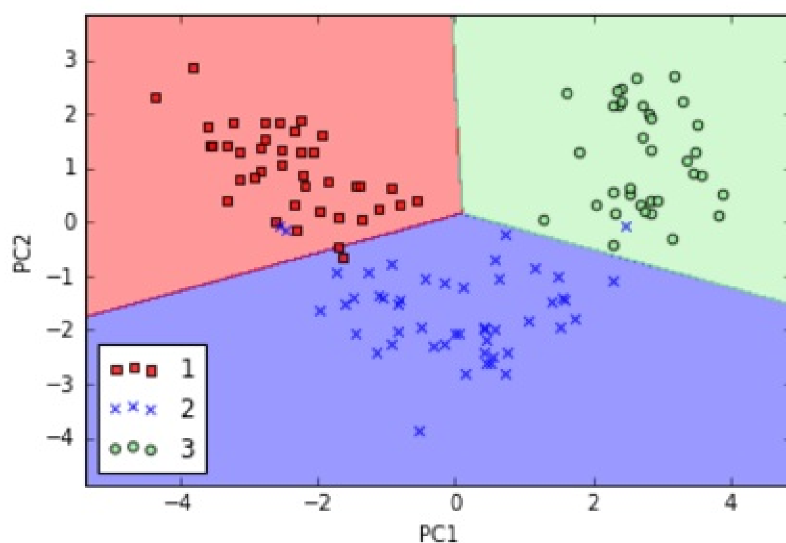
In [46]:
```
X_train_pca = pca.fit_transform(X_train_std)
X_test_pca = pca.transform(X_test_std)
```

In [47]:
```
lr.fit(X_train_pca, y_train)
```

Out[47]:
```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
          penalty='l2', random_state=None, solver='liblinear', tol=0.0001,
          verbose=0, warm_start=False)
```

In [48]:
```
plot_decision_regions(X_train_pca, y_train, classifier=lr)
plt.xlabel('PC1')
plt.ylabel('PC2')
plt.legend(loc='lower left')
plt.show()
```

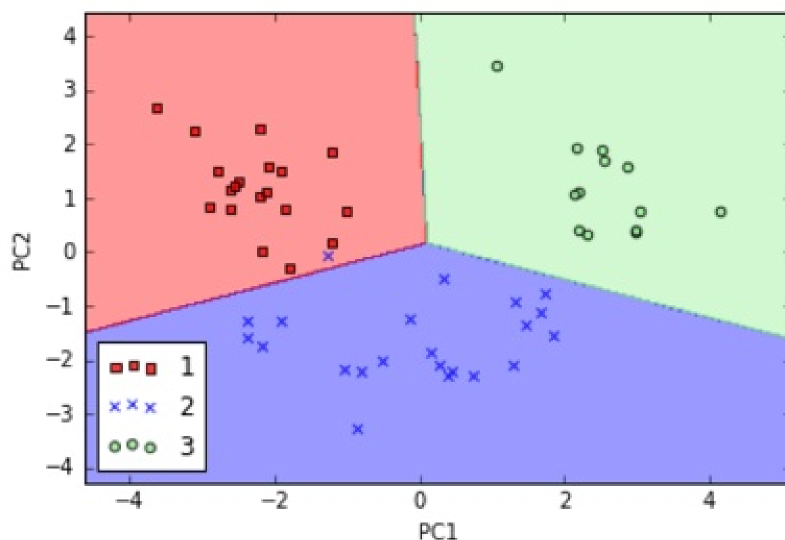
执行上面的代码，我们应该得到如下的决策界：



如果你仔细观察我们自己实现的PCA得到的散点图和调用sklearn中PCA得到的散点图，两幅图看起来是镜像关系！这是由于NumPy和sklearn求解特征向量时计算的差异，如果你实在看不惯，只需要将其中一个得到的特征向量*(-1)即可。还要注意特征向量一般都要归一化。

我们再看看决策界在测试集的分类效果：

```
In [50]: plot_decision_regions(X_test_pca, y_test, classifier=lr)
         plt.xlabel('PC1')
         plt.ylabel('PC2')
         plt.legend(loc='lower left')
         plt.show()
```



啊哈！测试集仅有一个样本被错误分类，效果很棒。

怎样用sklearn的PCA得到每个主成分(不是特征)的方差解释率呢？很简单，初始化PCA时，n*components*设置为*None*，然后通过*explained_variance_ratio*属性得到每个主成分的方差解释率：

```
In [51]: pca = PCA(n_components=None)

In [52]: X_train_pca = pca.fit_transform(X_train_std)

In [53]: pca.explained_variance_ratio_

Out[53]: array([ 0.37329648,  0.18818926,  0.10896791,  0.07724389,  0.06478595,
                 0.04592014,  0.03986936,  0.02521914,  0.02258181,  0.01830924,
                 0.01635336,  0.01284271,  0.00642076])
```

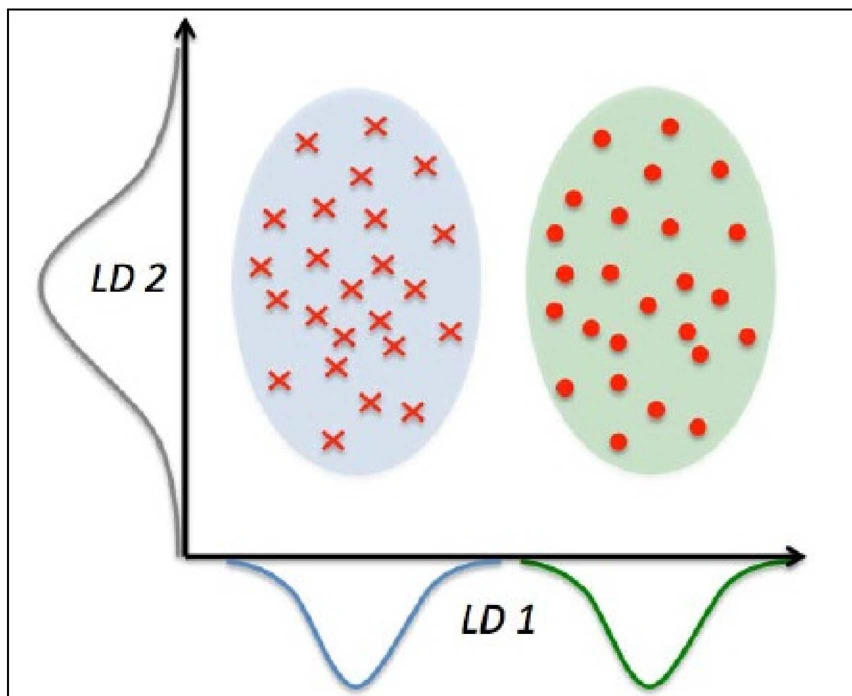
此时n_components=None, 我们并没有做降维。

LDA进行监督数据压缩

LDA(linear discriminant analysis, 线性判别分析)是另一种用于特征抽取的技术，它可以提高计算效率，对于非正则模型也能减小过拟合。

虽然LDA的很多概念和PCA很像，但他俩的目标不同，PCA目标是找到正交的主成分同时保持数据集的最大方差，LDA的目标是为每个类单独优化，得到各个类的最优特征子集。PCA和LDA都是线性转换技术，用于数据压缩，前者是无监算法，后者是监督算法。看到监督两个字，可能你会认为对于分类任务，LDA要比PCA效果更好，但实际却不是这样，在某些分类任务情境下，用PCA预处理数据得到的结果要比LDA号，比如，如果每个类含有的样本比较少。

下图画出了对于二分类问题，LDA的一些概念：



x轴的线性判别(LD 1),很好地将两个正态分布的类别数据分离。虽然y轴的线性判别(LD 2)捕捉了数据集中的大量方差，但却不是一个好的线性判别，因为它没有捕捉任何与类别判别相关的信息。

LDA 的一个假设是数据服从正态分布。同时，我们也假设各个类含有相同的协方差矩阵，每个特征都统计独立。即使真实数据可能不服从上面的某几个假设，但LDA依然具有很好的表现。

在学习LDA内部原理前，我们先将它的几大步骤列出来：

-

1. 将d维度原始数据进行标准化.

- 1. 对每一个类，计算d维度的平均向量.

- 1. 构建类间(between-class)散点矩阵S_B和类内(within-class)散点矩阵S_W.

- 1. 计算矩阵$S_W^{-1}S_B$的特征向量和特征值.

- 1. 选择值最大的前k个特征值对应的特征向量，构建d*d维度的转换矩阵W,每一个特征向量是W的一列.

- 1. 使用矩阵W将原始数据集映射到新的特征子空间.

Note 我们在应用LDA时做出的假设是：特征服从正态分布并且彼此独立，每个类的协方差矩阵都相同。现实中的数据当然不可能真的全部服从这些假设，但是不用担心，即使某一个甚至多个假设不成立，LDA也有不俗的表现.(R.O.Duda, P.E. Hart, and D.G.Stork. *Pattern Classification*. 2nd.Edition. New York, 2001).

计算散点矩阵

数据标准化前面已经说过了，这里就不再讲了，我们说一下如何计算平均向量，然后用这些平均向量分别构建类内散点矩阵和类间散点矩阵。

每一个平均向量m_i存储了类别i的样本的平均特征值u_m:

$$\boldsymbol{m}_i = \frac{1}{n_i}\sum_{x \in D_i}^{c} \boldsymbol{x}_m$$

Wine数据集有三个类，每一个的平均向量：

$$\boldsymbol{m}_i = \begin{bmatrix} \mu_{i,alcohol} \\ \mu_{i,malic\ acid} \\ \vdots \\ \mu_{i,proline} \end{bmatrix} \quad i \in \{1,2,3\}$$

In [54]:
```
np.set_printoptions(precision=4)
```

In [55]:
```
mean_vecs = []
```

In [56]:
```
for label in range(1, 4):
    mean_vecs.append(np.mean(X_train_std[y_train == label], axis=0))
    print('MV %s: %s\n' % (label, mean_vecs[label-1]))
```

```
MV 1: [ 0.9259 -0.3091  0.2592 -0.7989  0.3039  0.9608  1.0515 -0.6306  0.5354
  0.2209  0.4855  0.798   1.2017]

MV 2: [-0.8727 -0.3854 -0.4437  0.2481 -0.2409 -0.1059  0.0187 -0.0164  0.1095
 -0.8796  0.4392  0.2776 -0.7016]

MV 3: [ 0.1637  0.8929  0.3249  0.5658 -0.01   -0.9499 -1.228   0.7436 -0.7652
  0.979  -1.1698 -1.3007 -0.3912]
```

有了平均向量，我们就可以计算类内散点矩阵S_W:

$$S_W = \sum_{i=1}^{c} S_i$$

S_W就是每个类的散点矩阵S_i总和。

$$S_i = \sum_{x \in D_i}^{c} (x - m_i)(x - m_i)^T$$

代码：

In [57]:
```
d = 13 #特征维度
```

In [58]:
```
S_W = np.zeros((d, d))
```

In [59]:
```
for label, mv in zip(range(1, 4), mean_vecs):
    class_scatter = np.zeros((d, d))
    for row in X[y==label]:
        row, mv = row.reshape(d, 1), mv.reshape(d, 1)
        class_scatter += (row-mv).dot((row-mv).T)
    S_W += class_scatter

print('Within-class scatter matrix: %sx%s' %(S_W.shape[0], S_W.shape[1]))
```

```
Within-class scatter matrix: 13x13
```

当我们计算散点矩阵时，我们做出的假设是训练集中的类别时均匀分布的。实际情况往往并不是这样，比如我们将Wine训练集各个类别个数打印出来：

In [60]:
```
print('Class label distribution: %s' %np.bincount(y_train)[1:])
```

```
Class label distribution: [40 49 35]
```

所以在得到每个类别的散点矩阵S_i后，我们要将其缩放，然后再相加得到S_W。如果我们将散点矩阵S_i除以各个类别内样本数N_i,我们实际上是在计算协方差矩阵□. 协方差矩阵是散点矩阵的归一化结果：

$$\Sigma_i = \frac{1}{N_i} S_W = \frac{1}{N_i} \sum_{x \in D_i}^{c} (x - m_i)(x - m_i)^T$$

In [61]:
```
d = 13 #特征维度
```

In [62]:
```
S_W = np.zeros((d, d))
```

In [63]:
```
for label, mv in zip(range(1, 4), mean_vecs):
    class_scatter = np.cov(X_train_std[y_train==label].T)
    S_W += class_scatter
print('Scaled within-class scatter matrix: %sx%s' %(S_W.shape[0], S_W.shape[1]))
```

```
Scaled within-class scatter matrix: 13x13
```

得到缩放后的类内散点矩阵后，我们接下来计算类间散点矩阵S_B:

$$S_B = \sum_{i=1}^{c} N_i (m_i - m)(m_i - m)^T$$

其中，m是整个训练集所有样本的特征平均值.

In [88]:
```
mean_overall = np.mean(X_train_std, axis=0)
```

In [89]:
```
d = 13
```

In [90]:
```
S_B = np.zeros((d, d))
```

In [91]:
```
for i, mean_vec in enumerate(mean_vecs):
    n = X[y==i+1, :].shape[0]
    mean_vec = mean_vec.reshape(d, 1)
    mean_overall = mean_overall.reshape(d, 1)
    S_B += n* (mean_vec - mean_overall).dot((mean_vec - mean_overall).T)
```

In [92]:
```
print('Between-class scatter matrix: %sx%s' %(S_B.shape[0], S_B.shape[1]))
```

```
Between-class scatter matrix: 13x13
```

为特征子空间选择线性判别式

剩下的LDA步骤和PCA很像了。不同点是PCA分解协方差矩阵得到特征值和特征向量，LDA分解$S_W^{-1}S_B$得到特征值和特征向量：

In [93]:
```
eigen_vals, eigen_vecs = np.linalg.eig(np.linalg.inv(S_W).dot(S_B))
```

得到特征值后，对其降序排序：

In [94]:
```
eigen_pairs = [(np.abs(eigen_vals[i]), eigen_vecs[:, i])
               for i in range(len(eigen_vals))]
```

In [95]:
```
eigen_pairs = sorted(eigen_pairs, key=lambda k: k[0], reverse=True)
```

In [96]:
```
print('Eigenvalus in decreasing order: \n')
for eigen_val in eigen_pairs:
    print(eigen_val[0])
```

```
Eigenvalus in decreasing order:

643.015384346
225.086981854
7.85251171806e-14
3.78379173225e-14
3.40354823881e-14
3.40354823881e-14
2.84217094304e-14
1.62777129848e-14
1.62777129848e-14
6.4333539528e-15
6.4333539528e-15
5.5201562294e-15
2.1107086452e-15
```

我们可以看到只有两个特征值不为0，其余11个特征值实质是0，这里只不过由于计算机浮点表示才不为0。极端情况是特征向量共线，此时协方差矩阵秩为1，只有一个非0特征值。

为了度量线性判别式(特征向量)捕捉到了多少的类判别信息，我们画出类似方差解释率的线性判别图：

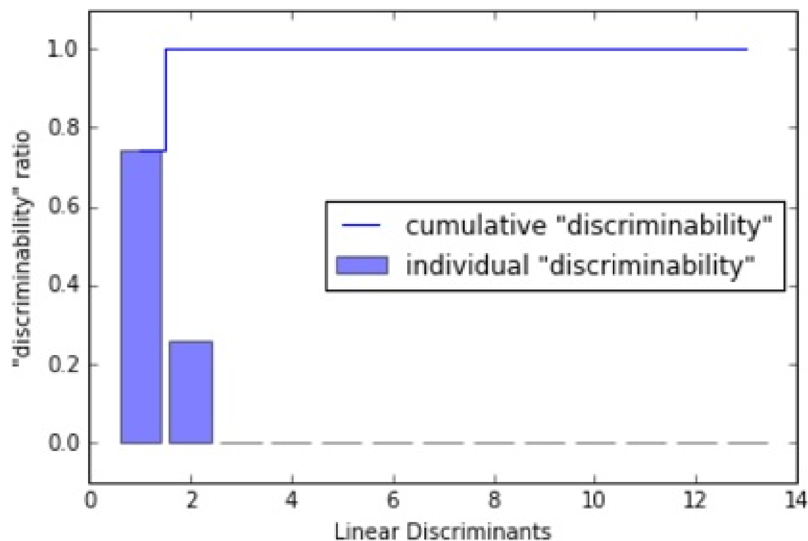
```
In [97]: tot = sum(eigen_vals.real)
```

```
In [98]: discr = [(i/tot) for i in sorted(eigen_vals.real, reverse=True)]
```

```
In [99]: cum_discr = np.cumsum(discr)
```

```
In [100]: plt.bar(range(1, 14), discr, alpha=0.5, align='center', label='individual "discriminability"')
          plt.step(range(1, 14), cum_discr, where='mid', label='cumulative "discriminability"')
          plt.ylabel('"discriminability" ratio')
          plt.xlabel('Linear Discriminants')
          plt.ylim([-0.1, 1.1])
          plt.legend(loc='best')
          plt.show()
```

我们可以看到前两个线性判别捕捉到了Wine训练集中100%的有用信息：



然后由这两个线性判别式来创建转换矩阵W:

```
In [101]: w = np.hstack((eigen_pairs[0][1][:, np.newaxis].real,
                         eigen_pairs[1][1][:, np.newaxis].real))
```

```
In [102]: print('Matrix W:\n', w)

('Matrix W:\n', array([[-0.0707, -0.3778],
       [ 0.0359, -0.2223],
       [-0.0263, -0.3813],
       [ 0.1875,  0.2955],
       [-0.0033,  0.0143],
       [ 0.2328,  0.0151],
       [-0.7719,  0.2149],
       [-0.0803,  0.0726],
       [ 0.0896,  0.1767],
       [ 0.1815, -0.2909],
       [-0.0631,  0.2376],
       [-0.3794,  0.0867],
       [-0.3355, -0.586 ]]))
```

原始数据映射到新特征空间

有了转换矩阵W,我们就可以将原始数据映射到新的特征空间了：

$$X' = XW$$

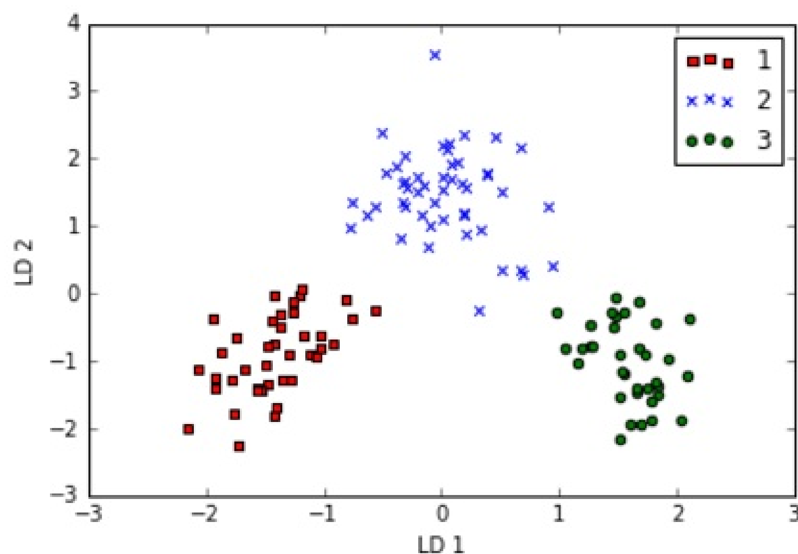
In [103]:
```
X_train_lda = X_train_std.dot(w)
```

In [104]:
```
colors = ['r', 'b', 'g']
```

In [105]:
```
markers = ['s', 'x', 'o']
```

In [107]:
```
for l, c, m in zip(np.unique(y_train), colors, markers):
    plt.scatter(X_train_lda[y_train==l, 0],
                X_train_lda[y_train==l, 1],
                c=c, label=l, marker=m)
plt.xlabel('LD 1')
plt.ylabel('LD 2')
plt.legend(loc='upper right')
plt.show()
```

新数据集，明显线性可分：



调用sklearn中LDA

我们通过一步步地实现LDA来加深理解，现在看看sklearn中如何使用现成的LDA：

```
In [108]: from sklearn.lda import LDA
```

```
c:\python27\lib\site-packages\sklearn\lda.py:4: DeprecationWarning: lc
is in 0.17 and will be removed in 0.19
  "in 0.17 and will be removed in 0.19", DeprecationWarning)
```

```
In [109]: lda = LDA(n_components=2)
```

```
In [110]: X_train_lda = lda.fit_transform(X_train_std, y_train)
```

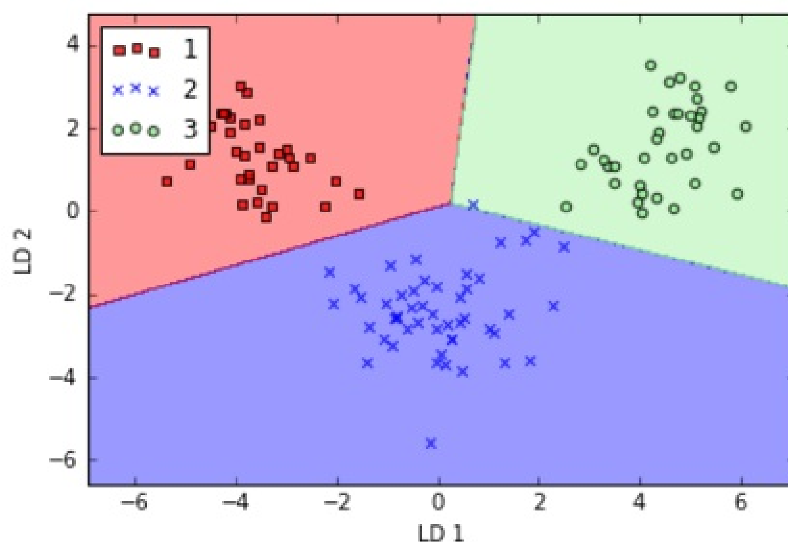
逻辑斯蒂回归建模

```
In [111]: lr = LogisticRegression()
```

```
In [112]: lr = lr.fit(X_train_lda, y_train)
```

```
In [113]: plot_decision_regions(X_train_lda, y_train, classifier=lr)
          plt.xlabel('LD 1')
          plt.ylabel('LD 2')
          plt.legend(loc='upper left')
          plt.show()
```

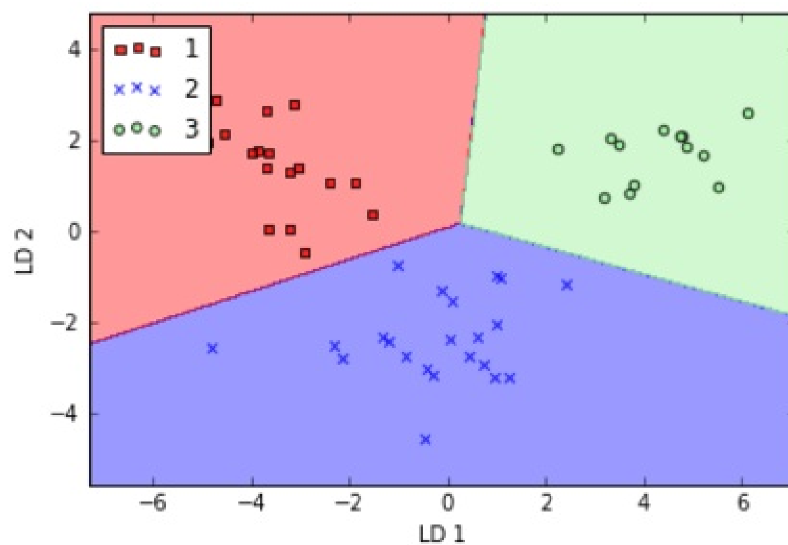
有一个样本被错分类：



如果降低正则项的影响，完全正确分类训练集。当然了，过拟合并没有什么好处。我们看一下现在模型对测试集的分类效果：

```
In [115]: X_test_lda = lda.transform(X_test_std)
```

```
In [116]: plot_decision_regions(X_test_lda, y_test, classifier=lr)
          plt.xlabel('LD 1')
          plt.ylabel('LD 2')
          plt.legend(loc='upper left')
          plt.show()
```

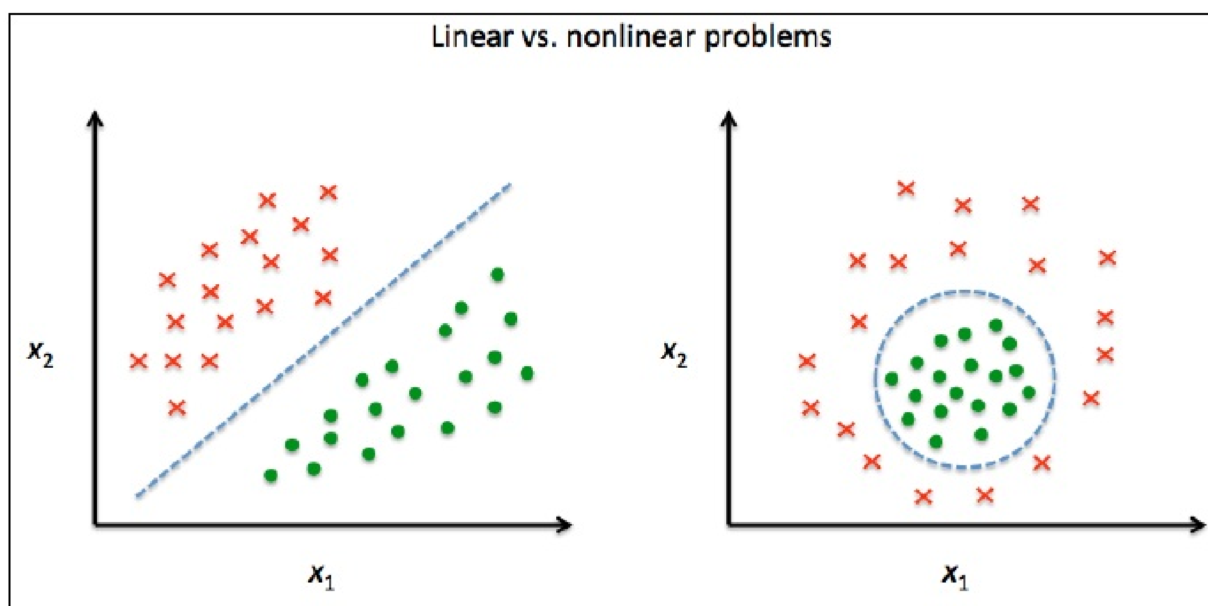


Wow！100%的准确率。

使用核PCA进行非线性映射

许多机器学习算法都有一个假设：输入数据要是线性可分的。感知机算法必须针对完全线性可分数据才能收敛。考虑到噪音，Adalien、逻辑斯蒂回归和SVM并不会要求数据完全线性可分。

但是现实生活中有大量的非线性数据，此时用于降维的线性转换手段比如PCA和LDA效果就不会太好。这一节我们学习PCA的核化版本，核PCA。这里的"核"与核SVM相近。 运用核PCA，我们能将非线性可分的数据转换到新的、低维度的特征子空间，然后运用线性分类器解决。



核函数和核技巧

还记得在核SVM那里，我们讲过解决非线性问题的手段是将他们映射到新的高维特征空间，此时数据在高维空间线性可分。为了将数据映射到高维k空间，我们定义了非线性映射函数:

$$\phi:\mathbb{R}^d \rightarrow \mathbb{R}^k \quad (k >> d)$$

我们可以把核函数的功能理解为：通过创造出原始特征的一些非线性组合，然后将原来的d维度数据集映射到k维度特征空间，d<k。举个例子，特征向量，x是列向量包含d个特征，d=2，可以按照如下的规则将其映射到3维度特征空间：

$$\boldsymbol{x} = [x_1, x_2]^T$$

$$\downarrow \phi$$

$$\mathbf{z} = \left[x_1^2, \sqrt{2x_1x_2}, x_2^2\right]^T$$

同理核PCA的工作机制：通过核PCA的非线性映射，将数据转换到一个高维度空间，然后在这个高维度空间运用标准PCA重新将数据映射到一个比原来还低的空间，最后就可以用线性分类器解决问题了。不过，这种方法涉及到两次映射转换，计算成本非常高，由此引出了核技巧(kernel trick)。

使用核技巧，我们能在原始特征空间直接计算两个高维特征向量的相似性(不需要先特征映射，再计算相似性)。

在介绍核技巧前，我们先回顾标准PCA的做法。我们按照如下公式计算两个特征k和j的协方差：

$$\sigma_{jk} = \frac{1}{n}\sum_{i=1}^{n}\left(x_j^{(i)} - \mu_j\right)\left(x_k^{(i)} - \mu_k\right)$$

由于我们对数据已做过标准化处理，特征平均值为0，上式等价于：

$$\sigma_{jk} = \frac{1}{n}\sum_{i=1}^{n} x_j^{(i)} x_k^{(i)}$$

同样，我们能够得到协方差矩阵：

$$\Sigma = \frac{1}{n}\sum_{i=1}^{n} \boldsymbol{x}^{(i)} \boldsymbol{x}^{(i)T}$$

Bernhard Scholkopf(B. Scholkopf, A.Smola, and K.R. Muller. *Kernel Principal Component Analysis*. pages 583-588, 1997)得到了上式的泛化形式,用非线性特征组合替换原始数据集两个样本之间的点乘：

$$\sum = \frac{1}{n}\sum_{i=1}^{n}\phi\left(\boldsymbol{x}^{(i)}\right)\phi(\boldsymbol{x}^{(i)})^{T}$$

为了从协方差矩阵中得到特征向量(主成分)，我们必须求解下面的等式：

$$\Sigma \boldsymbol{v} = \lambda \boldsymbol{v}$$

$$\Rightarrow \frac{1}{n}\sum_{i=1}^{n}\phi\left(\boldsymbol{x}^{(i)}\right)\phi\left(\boldsymbol{x}^{(i)}\right)^{T}\boldsymbol{v} = \lambda \boldsymbol{v}$$

$$\Rightarrow v = \frac{1}{n\lambda}\sum_{i=1}^{n}\phi\left(\boldsymbol{x}^{(i)}\right)\phi\left(\boldsymbol{x}^{(i)}\right)^{T}\boldsymbol{v} = \frac{1}{n}\sum_{i=1}^{n}\boldsymbol{a}^{(i)}\phi\left(\boldsymbol{x}^{(i)}\right)$$

其中，是协方差矩阵的特征值和特征向量，的求法见下面几段内容。

我们求解核矩阵：

首先，我们写出协方差矩阵的矩阵形式，是一个n*k的矩阵：

$$\sum = \frac{1}{n}\sum_{i=1}^{n}\phi\left(\boldsymbol{x}^{(i)}\right)\phi\left(\boldsymbol{x}^{(i)}\right)^{T} = \frac{1}{n}\phi(\boldsymbol{X})^{T}\phi(\boldsymbol{X})$$

我们将特征向量写作：

$$v = \frac{1}{n}\sum_{i=1}^{n}a^{(i)}\phi\left(\boldsymbol{x}^{(i)}\right) = \lambda\phi(\boldsymbol{X})^{T}\boldsymbol{a}$$

由于，得：

$$\frac{1}{n}\phi(\boldsymbol{X})^{T}\phi(\boldsymbol{X})\phi(\boldsymbol{X})^{T}\boldsymbol{a} = \lambda\phi(\boldsymbol{X})^{T}\boldsymbol{a}$$

等式两边左乘：

$$\frac{1}{n}\phi(\boldsymbol{X})\phi(\boldsymbol{X})^T\phi(\boldsymbol{X})\phi(\boldsymbol{X})^T\boldsymbol{a}=\lambda\phi(\boldsymbol{X})\phi(\boldsymbol{X})^T\boldsymbol{a}$$

$$\Rightarrow\frac{1}{n}\phi(\boldsymbol{X})\phi(\boldsymbol{X})^T\boldsymbol{a}=\lambda\boldsymbol{a}$$

$$\Rightarrow\frac{1}{n}\boldsymbol{K}\boldsymbol{a}=\lambda\boldsymbol{a}$$

这里的就是相似性(核)矩阵：

$$\boldsymbol{K}=\phi(\boldsymbol{X})\phi(\boldsymbol{X})^T$$

回忆核SVM我们使用核技巧避免了直接计算：

$$k\left(\boldsymbol{x}^{(i)},\boldsymbol{x}^{(j)}\right)=\phi\left(\boldsymbol{x}^{(i)}\right)^T\phi\left(\boldsymbol{x}^{(j)}\right)$$

核PCA同样不需要像标准PCA那样构建转换矩阵，我们使用核函数代替计算。所以，你可以把核函数(简称，核)理解为计算两个向量点乘的函数，结果可看做两个向量的相似度。

常用的核函数有：

- 多项式核：

$$k\left(\boldsymbol{x}^{(i)},\boldsymbol{x}^{(j)}\right)=\left(\boldsymbol{x}^{(i)T}\boldsymbol{x}^{(j)}+\theta\right)^p$$

，是阈值，p是由用户设定的指数。

- 双曲正切(sigmoid)核：

$$k\left(\boldsymbol{x}^{(i)},\boldsymbol{x}^{(j)}\right)=\tanh\left(\eta\boldsymbol{x}^{(i)T}\boldsymbol{x}^{(j)}+\theta\right)$$

- 径向基函数核(高斯核)：

$$k\left(\boldsymbol{x}^{(i)},\boldsymbol{x}^{(j)}\right)=\exp\left(-\frac{\left\|\boldsymbol{x}^{(i)}-\boldsymbol{x}^{(j)}\right\|^{2}}{2\sigma^{2}}\right)$$

It is also written as follows:

$$k\left(\boldsymbol{x}^{(i)},\boldsymbol{x}^{(j)}\right)=\exp\left(-\gamma\left\|\boldsymbol{x}^{(i)}-\boldsymbol{x}^{(j)}\right\|^{2}\right)$$

现在总结一下核PCA的步骤，以RBF核为例：

1 计算核(相似)矩阵k，也就是计算任意两个训练样本：

$$k\left(\boldsymbol{x}^{(i)},\boldsymbol{x}^{(j)}\right)=\exp\left(-\gamma\left\|\boldsymbol{x}^{(i)}-\boldsymbol{x}^{(j)}\right\|^{2}\right)$$

得到K:

$$\boldsymbol{K}=\begin{bmatrix}\kappa\left(\boldsymbol{x}^{(1)},\boldsymbol{x}^{(1)}\right) & \kappa\left(\boldsymbol{x}^{(1)},\boldsymbol{x}^{(2)}\right) & \cdots & \kappa\left(\boldsymbol{x}^{(1)},\boldsymbol{x}^{(n)}\right)\\ \kappa\left(\boldsymbol{x}^{(2)},\boldsymbol{x}^{(1)}\right) & \left(\boldsymbol{x}^{(2)},\boldsymbol{x}^{(2)}\right) & \cdots & \kappa\left(\boldsymbol{x}^{(2)},\boldsymbol{x}^{(n)}\right)\\ \vdots & \vdots & \ddots & \vdots\\ \kappa\left(\boldsymbol{x}^{(n)},\boldsymbol{x}^{(1)}\right) & \kappa\left(\boldsymbol{x}^{(d)},\boldsymbol{x}^{(2)}\right) & \cdots & \kappa\left(\boldsymbol{x}^{(n)},\boldsymbol{x}^{(n)}\right)\end{bmatrix}$$

举个例子，如训练集有100个样本，则对称核矩阵K的维度是100*100。

2 对核矩阵K进行中心化处理：

$$\boldsymbol{K}'=\boldsymbol{K}-1_{n}\boldsymbol{K}-\boldsymbol{K}1_{n}+1_{n}\boldsymbol{K}1_{n}$$

其中,1_n是n*n的矩阵，n=训练集样本数，1_n中每个元素都等于□.

3 计算 的特征值，取最大的k个特征值对应的特征向量。不同于标准PCA，这里的特征向量并不是主成分轴。

第2步为什么要计算 ? 因为在PCA我们总是处理标准化的数据，也就是特征的平均值为0。当我们用非线性特征组合 替代点乘时，我们并没有显示计算新的特征空间也就没有在新特征空间做标准化处理，我们不能保证新特征空间下的特征平均值为0，所以要对K做中心化。

用Python实现核PCA

上一节我们讨论了核PCA的原理。现在我们根据上一节的三个步骤，自己实现一个核PCA。借助SciPy和NumPy，其实实现核PCA很简单：

In [85]:
```
from scipy.spatial.distance import pdist, squareform
```

In [86]:
```
from scipy import exp
```

In [87]:
```
from scipy.linalg import eigh
```

In [88]:
```
import numpy as np
```

In [89]:
```
def rbf_kernel_pca(X, gamma, n_components):
    """
    RBF kernel PCA implementation.

    X: shape=[n_samples, n_features]
    gamma: float, Tuning parameter of the RBF kernel

    n_components: int. Number of pricipal components to return
    """
    #Calculate pariwise squared Euclidean distances
    sq_dists = pdist(X, 'sqeuclidean')

    #Convert pairwise distances into a square matrix
    mat_sq_dists = squareform(sq_dists)

    #Compute the symmetric kernel matrix.
    K = exp(-gamma * mat_sq_dists)

    #Center the kernel matrix
    N = K.shape[0]
    one_n = np.ones((N, N)) / N
    K = K - one_n.dot(K) - K.dot(one_n) + one_n.dot(K).dot(one_n)

    #Obtaining eigenparis from the K'
    eigvals, eigvecs = eigh(K)

    # Collect the top k eigenvectors(projected samples)
    X_pc = np.column_stack((eigvecs[:, -i] for i in range(1, n_components + 1)))

    return X_pc
```

RBF核PCA的一个缺点是需要人工设置□值，调参不易。第六章我们会介绍调参技巧。

例1 半月形数据分割

现在我们创建一个非线性数据集试一下rbf_kernel_pca的效果，数据集100个样本，每个样本两维度特征，数据分布呈两个半月形：

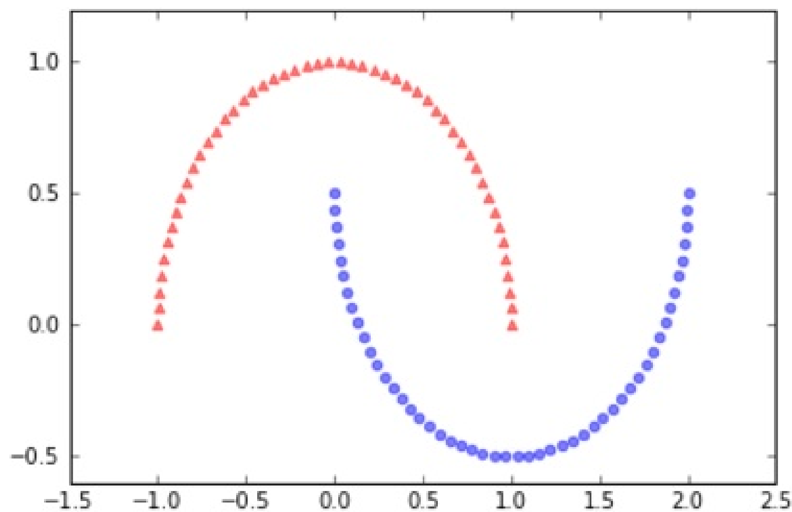
In [84]:
```
from sklearn.datasets import make_moons
```

In [85]:
```
X, y = make_moons(n_samples=100, random_state=123)
```

In [86]:
```
plt.scatter(X[y==0, 0], X[y==0, 1],
            color='red', marker='^', alpha=0.5)

plt.scatter(X[y==1, 0], X[y==1, 1],
            color='blue', marker='o', alpha=0.5)

plt.show()
```



每个半月数据是一类。

显然图中的数据不是线性可分的，我们的目标是通过核PCA将"曲线"映射到"直线"上，然后就可以用线性分类器了。

我们先看一下如果用标准PCA处理，会得到什么结果：

In [87]:
```
from sklearn.decomposition import PCA
```

In [88]:
```
scikit_pca = PCA(n_components=2)
```

In [89]:
```
X_spca = scikit_pca.fit_transform(X)
```

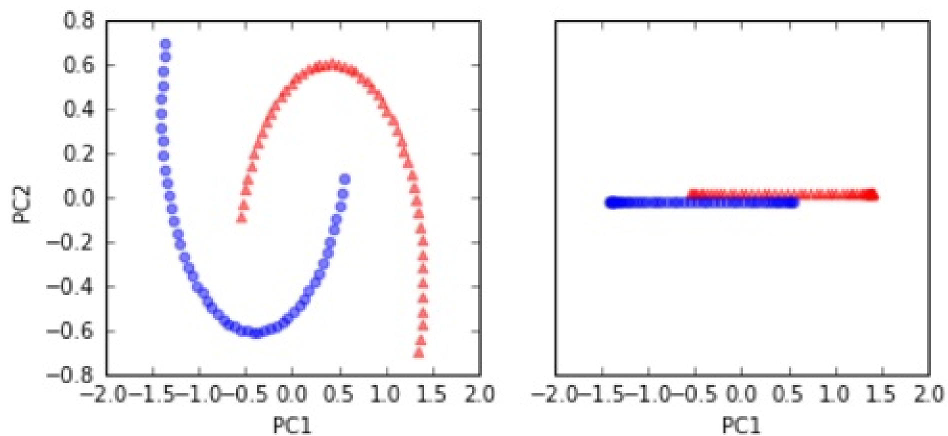
In [90]:
```
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(7,3))

ax[0].scatter(X_spca[y==0, 0], X_spca[y==0, 1],
              color='red', marker='^', alpha=0.5)

ax[0].scatter(X_spca[y==1, 0], X_spca[y==1, 1],
              color='blue', marker='o', alpha=0.5)

ax[1].scatter(X_spca[y==0, 0], np.zeros((50, 1))+0.02,
              color='red', marker='^', alpha=0.5)
ax[1].scatter(X_spca[y==1, 0], np.zeros((50, 1))-0.02,
              color='blue', marker='o', alpha=0.5)

ax[0].set_xlabel('PC1')
ax[0].set_ylabel('PC2')
ax[1].set_ylim([-1, 1])
ax[1].set_yticks([])
ax[1].set_xlabel('PC1')
plt.show()
```



很显然，用标准PCA降维后的数据无法用线性分类器处理。还要注意右边子图中的蓝线我们故意下调了一点点，红线上调了一点点，这是为了清楚地观察他们的重合部分。

现在我们尝试rbf_kernel_pca，看看能不能解决非线性数据：

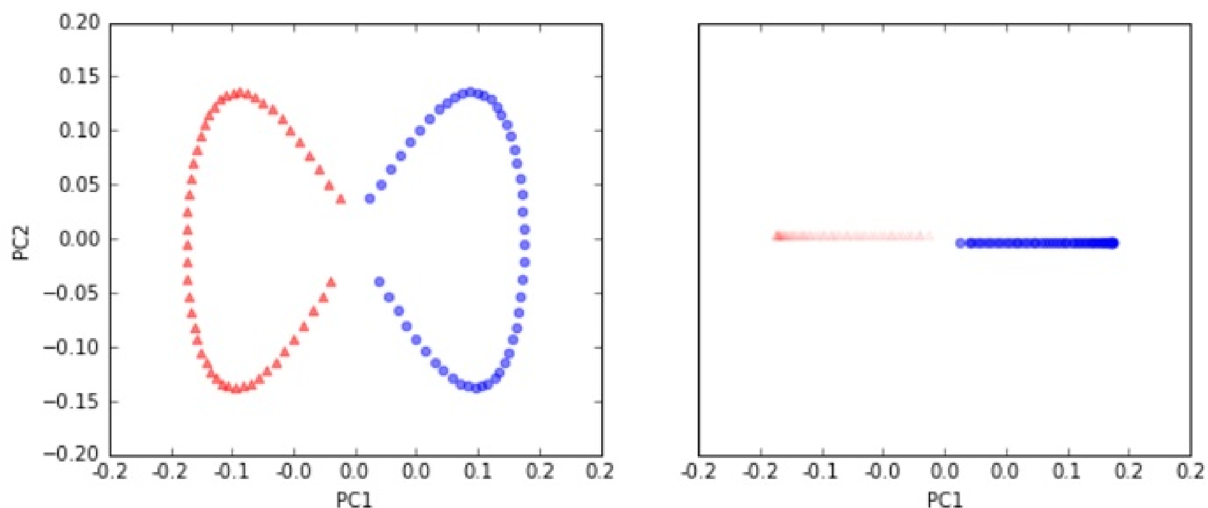
In [91]:
```
from matplotlib.ticker import FormatStrFormatter
```

In [92]:
```
X_kpca = rbf_kernel_pca(X, gamma=15, n_components=2)
```

In [96]:
```
flg, ax = plt.subplots(nrows=1, ncols=2, figsize=(10, 4))
ax[0].scatter(X_kpca[y==0, 0], X_kpca[y==0, 1],
              color='red', marker='^', alpha=0.5)
ax[0].scatter(X_kpca[y==1, 0], X_kpca[y==1, 1],
              color='blue', marker='o', alpha=0.5)

ax[1].scatter(X_kpca[y==0, 0], np.zeros((50,1))+0.02,
              color='red', marker='^',alpha=0.05)
ax[1].scatter(X_kpca[y==1, 0], np.zeros((50,1))-0.02,
              color='blue',marker='o',alpha=0.5)
ax[0].set_xlabel('PC1')
ax[0].set_ylabel('PC2')
ax[1].set_ylim([-1, 1])
ax[1].set_yticks([])
ax[1].set_xlabel('PC1')
ax[0].xaxis.set_major_formatter(FormatStrFormatter('%0.1f'))
ax[1].xaxis.set_major_formatter(FormatStrFormatter('%0.1f'))
plt.show()
```

我们可以看到经过核PCA处理后的数据线性可分了，所以可以直接用线性分类器来训练一个合适的模型。



RBF核PCA的缺点就是需要人工设定，这个值不是凭空瞎想的，而是要做实验确定，在第六章，我们会讨论调参的技巧。

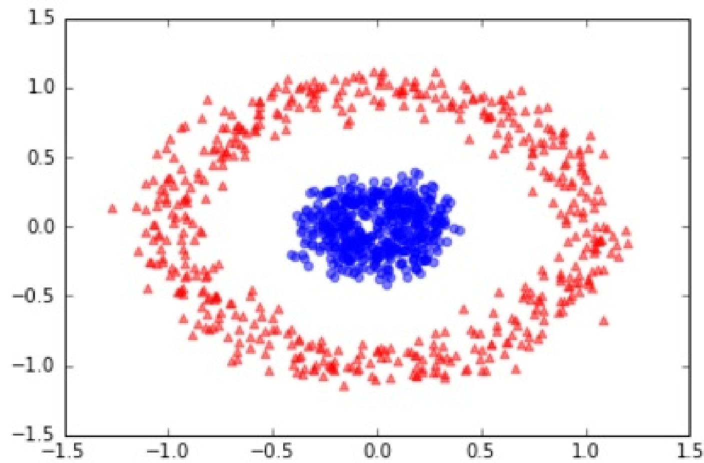
例2 分离同心圆数据

上一节我们创建了一个半圆形的数据，然后可以用核PCA线性分离，我们再来看另一个有趣的非线性数据集：

```
In [97]: from sklearn.datasets import make_circles
```

```
In [98]: X, y = make_circles(n_samples=1000, random_state=123, noise=0.1, factor=0.2)
```

```
In [99]: plt.scatter(X[y==0, 0], X[y==0, 1],
                     color='red', marker='^', alpha=0.5)
         plt.scatter(X[y==1, 0], X[y==1, 1],
                     color='blue', marker='o', alpha=0.5)
         plt.show()
```



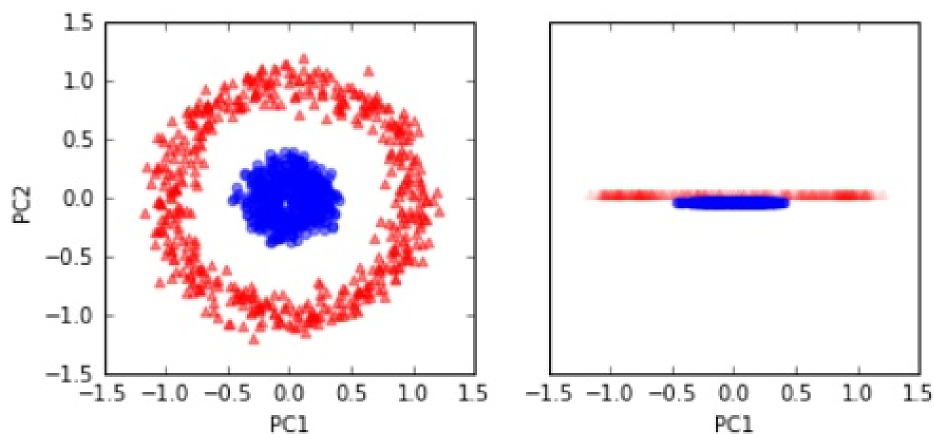
同样这也是一个二分类问题。我们分别用标准PCA和RBF核PCA处理数据集，然后比较他们的结果：

```
In [100]: scikit_pca = PCA(n_components=2)
```

```
In [101]: X_spca = scikit_pca.fit_transform(X)
```

```
In [102]: fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(7,3))
          ax[0].scatter(X_spca[y==0, 0], X_spca[y==0, 1],
                        color='red', marker='^', alpha=0.5)
          ax[0].scatter(X_spca[y==1, 0], X_spca[y==1, 1],
                        color='blue', marker='o', alpha=0.5)

          ax[1].scatter(X_spca[y==0, 0], np.zeros((500,1))+0.02,
                        color='red', marker='^',alpha=0.05)
          ax[1].scatter(X_spca[y==1, 0], np.zeros((500,1))-0.02,
                        color='blue',marker='o',alpha=0.5)
          ax[0].set_xlabel('PC1')
          ax[0].set_ylabel('PC2')
          ax[1].set_ylim([-1, 1])
          ax[1].set_yticks([])
          ax[1].set_xlabel('PC1')
          plt.show()
```



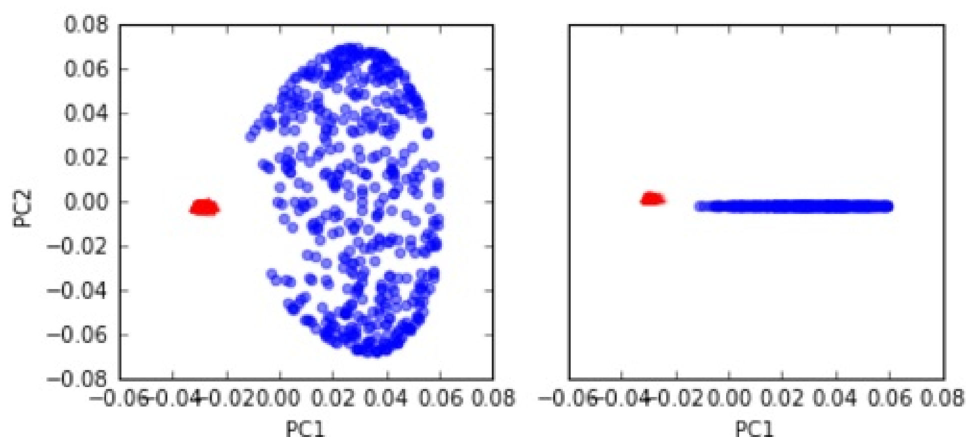
标准PCA效果不好，处理后不能应用线性分类器。

再来看看RBF核PCA的效果能不能让我们满意：

```
In [103]: X_kpca = rbf_kernel_pca(X, gamma=15, n_components=2)
```

```
In [104]: fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(7, 3))
          ax[0].scatter(X_kpca[y==0, 0], X_kpca[y==0, 1],
                        color='red', marker='^', alpha=0.5)
          ax[0].scatter(X_kpca[y==1, 0], X_kpca[y==1, 1],
                        color='blue', marker='o', alpha=0.5)

          ax[1].scatter(X_kpca[y==0, 0], np.zeros((500,1))+0.02,
                        color='red', marker='^',alpha=0.05)
          ax[1].scatter(X_kpca[y==1, 0], np.zeros((500,1))-0.02,
                        color='blue',marker='o',alpha=0.5)
          ax[0].set_xlabel('PC1')
          ax[0].set_ylabel('PC2')
          ax[1].set_ylim([-1, 1])
          ax[1].set_yticks([])
          ax[1].set_xlabel('PC1')
          plt.show()
```



Wow，结果非常好。

映射新的数据点

在前面的两个例子中，我们将原始的数据集映射到新的特征空间。不过在实际应用中，我们常常需要将多个数据集转换，比如训练集和测试集，还有可能在训练好模型后，又收集到的新数据。在本节，你将学习如何将不属于训练集的数据进行映射。

还记得在标准PCA中，我们通过计算 转换矩阵*输入样本，得到映射后的数据。转换矩阵的每一列是我们从协方差矩阵中得到的k个特征向量。现在，如何将这种思路应用到核PCA？在核PCA中，我们得到的特征向量来自归一化的核矩阵(centered kernel matrix)，而不是协方差矩阵，这意味着样本已经被映射到主成分轴v.因此，如果我们要把一个新样本 映射到主成分轴，我们要按照下式:

$$\phi(\boldsymbol{x}')^T \boldsymbol{v}$$

上式怎么算？当然不好算，好在我们还有核技巧，所以可以避免直接计算 。

和标准PCA不同的是，核PCA是一种基于内存的方法，这是什么意思呢？意思是每次对新样本进行映射时就要用到所有的训练集。因为要计算训练集中每个样本和新样本x'之间的RBF核(相似度):

$$\phi(\boldsymbol{x}')^T \boldsymbol{v} = \sum_i a^{(i)} \phi(\boldsymbol{x}')^T \phi\left(\boldsymbol{x}^{(i)}\right)$$

$$= \sum_i a^{(i)} k\left(\boldsymbol{x}', \boldsymbol{x}^{(i)}\right)^T$$

其中，核矩阵 的特征向量 和特征值 满足条件: 。

计算每一个训练集样本和新样本的$k()$后，我们必须用特征值对特征向量做归一化。所以呢，我们要修改一下前面实现的RBF PCA，能够返回核矩阵的特征向量：

```
from scipy.spatial.distance import pdist, squareform
from scipy import exp
from scipy.linalg import eigh
import numpy as np

def rbf_kernel_pca(X, gamma, n_components):
    """
    RBF kernel PCA implementation.
    """
    # Calculate pairwise squared Eculidean distances
    sq_dists = pdist(X, 'sqeuclidean')

    mat_sq_dists = squareform(sq_dists)
    #Compute the symmetric kernel matrix
    K = exp(-gamma * mat_sq_dists)
    #Center the kernle matrix
    N = K.shape[0]
    one_n = np.ones((N, N)) / N
    K = K - one_n.dot(K) - K.dot(one_n) + one_n.dot(K).dot(one_n)

    #Obtaining eigenpairs from the centered kernel matrix
    eigvals, eigvecs = eigh(K)

    alphas = np.column_stack((eigvecs[:, -i]
                             for i in range(1,n_components+1)))

    #Collect the corresponding eigenvalues
    lambdas = [eigvals[-i] for i in range(1, n_components+1)]

    return alphas, lambdas
```

现在，我们创建一个新的半月形数据集，然后用更新过的核PCA将其映射到一维子空间：

```
In [101]: X, y = make_moons(n_samples=100, random_state=123)
```

```
In [102]: alphas, lambdas =rbf_kernel_pca(X, gamma=15, n_components=1)
```

为了检验对于新数据点的映射表现，我们假设第26个点时新数据点x'，我们的目标就是将这个新数据点进行映射:

```
In [103]: x_new = X[25]
```

```
In [104]: x_new
```

```
Out[104]: array([ 1.8713,  0.0093])
```

```
In [105]: x_proj = alphas[25]
```

```
In [106]: x_proj
```

```
Out[106]: array([ 0.0788])
```

```
In [107]: def project_x(x_new, X, gamma, alphas, lambdas):
              pair_dist = np.array([np.sum((x_new-row)**2) for row in X])
              k = np.exp(-gamma * pair_dist)
              return k.dot(alphas/lambdas)
```

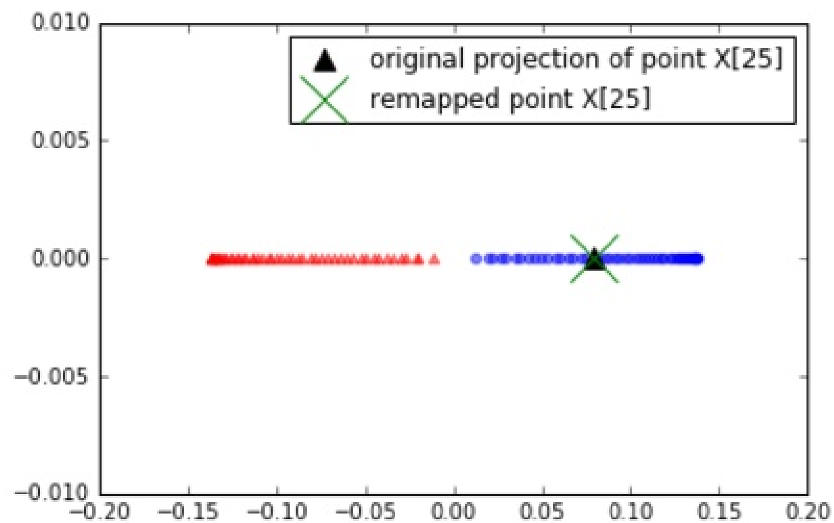
使用project_x函数，我们能够对新数据样本进行映射：

```
In [108]: x_reproj = project_x(x_new, X, gamma=15, alphas=alphas, lambdas=lambdas)
          x_reproj
```

```
Out[108]: array([ 0.0788])
```

最后，我们将第一主成分进行可视化：

In [109]:
```
plt.scatter(alphas[y==0, 0], np.zeros((50)),
            color = 'red', marker='^', alpha=0.5)
plt.scatter(alphas[y==1, 0], np.zeros((50)),
            color='blue',marker='o',alpha=0.5)
plt.scatter(x_proj, 0, color='black',
            label='original projection of point X[25]',
            marker='^', s=100)
plt.scatter(x_reproj, 0, color='green',
            label='remapped point X[25]',
            marker='x',s=500)
plt.legend(scatterpoints=1)
plt.show()
```



sklearn中的核PCA

sklearn.decomposition中有核PCA的实现，看看怎么用：

```
In [110]: from sklearn.decomposition import KernelPCA
```

```
In [111]: X, y = make_moons(n_samples=100, random_state=123)
```

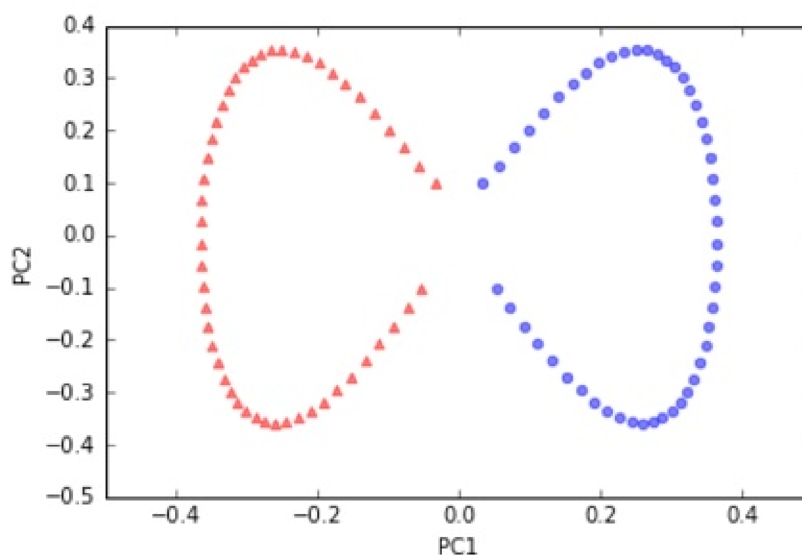
```
In [112]: scikit_kpca = KernelPCA(n_components=2,
                                  kernel='rbf', gamma=15)
```

```
In [115]: X_skernpca = scikit_kpca.fit_transform(X)
```

通过kernel参数设定不同的核函数。

将转换后的数据可视化：

```
In [116]: plt.scatter(X_skernpca[y==0, 0], X_skernpca[y==0, 1],
                      color='red', marker='^', alpha=0.5)
          plt.scatter(X_skernpca[y==1, 0], X_skernpca[y==1, 1],
                      color='blue', marker='o', alpha=0.5)
          plt.xlabel('PC1')
          plt.ylabel('PC2')
          plt.show()
```



总结

本章，你学习了三种基本的用于特征抽取的降维方法：标准PCA、LDA和核PCA。

使用PCA，我们将数据映射到一个低维度的子空间并且最大化正交特征轴的方差，PCA不考虑类别信息。LDA是一种监督降维方法，意味着他要考虑训练集的类别信息，目标是将类别最大化地可分。最后，你学习了核PCA，它能够将非线性数据集映射到低维特征空间，然后数据变成线性可分了。

第六章 模型评估和调参

前面几章，你学习了分类问题要用到的机器学习算法以及必备的数据预处理算法。现在，是时候学习如何训练一个好用的机器学习模型了，二者要涉及到两个重要的课题：模型评估和参数寻优。

本章，我们要学习：

- 获得对模型性能的无偏差估计
- 诊断常见的机器学习问题
- 模型调参
- 使用不同的性能评价指标对模型评估

通过管道创建工作流

当我们应用不同的预处理理技巧时，比如对特征标准化、对数据主成分分析，我们都需要重复利用某些参数，比如对训练集标准化后还要对测试集进行标准化(二者必须使用相同的参数)。

本节，你会学到一个非常有用的工具：管道(pipeline)，这里的管道不是Linux中的管道，而是sklearn中的Pipeline类，二者做的事情差不多。

读取Breast Cancer Wisconsin数据集

本章，我们要用到一个新的二分类数据集 **Breast Cancer Wisconsin**，它包含569个样本。每一条数据前两列是唯一的ID和相应的类别值(M=恶性肿瘤，B=良性肿瘤)，第3-32列是实数值的特征。

话不多说，先读取数据集，然后将y转为0，1：

```
In  [3]: df = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-databases/breast-cancer-wisconsin/wdbc.data', header=None)

In  [4]: from sklearn.preprocessing import LabelEncoder

In  [5]: X = df.iloc[:, 2:].values

In [12]: y = df.loc[:, 1].values

In [14]: np.unique(y)
Out[14]: array(['B', 'M'], dtype=object)

In [15]: le = LabelEncoder()

In [16]: y = le.fit_transform(y)

In [17]: np.unique(y)
Out[17]: array([0, 1], dtype=int64)
```

接着创建训练集和测试集:


```
In  [3]: df = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-databases/breast-cancer-wisconsin/wdbc.data', header=None)

In  [4]: from sklearn.preprocessing import LabelEncoder

In  [5]: X = df.iloc[:, 2:].values

In [12]: y = df.loc[:, 1].values

In [14]: np.unique(y)
Out[14]: array(['B', 'M'], dtype=object)

In [15]: le = LabelEncoder()

In [16]: y = le.fit_transform(y)

In [17]: np.unique(y)
Out[17]: array([0, 1], dtype=int64)
```

将transformer和Estimator放入同一个管道

前几章说过，很多机器学习算法要求特征取值范围要相同。因此，我们要对BCW数据集每一列做标准化处理，然后才能应用到线性分类器。此外，我们还想将原始的30维度特征压缩的2维度，这个交给PCA来做。

之前我们都是每一步执行一个操作，现在我们学习用管道将 StandardScaler, PCA和LogisticRegression连接起来：

```
In [20]: from sklearn.preprocessing import StandardScaler
         from sklearn.decomposition import PCA
         from sklearn.linear_model import LogisticRegression
         from sklearn.pipeline import Pipeline

In [21]: pipe_lr = Pipeline([('scl', StandardScaler()),
                             ('pca', PCA(n_components=2)),
                             ('clf', LogisticRegression(random_state=1))])

In [22]: pipe_lr.fit(X_train, y_train)

Out[22]: Pipeline(steps=[('scl', StandardScaler(copy=True, with_mean=True, with_std=True)), ('pca', PCA(copy=True, n_components=2, whiten=False)),
          ('clf', LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
                   penalty='l2', random_state=1, solver='liblinear', tol=0.0001,
                   verbose=0, warm_start=False))])

In [23]: print('Test Accuracy: %.3f' %pipe_lr.score(X_test, y_test))

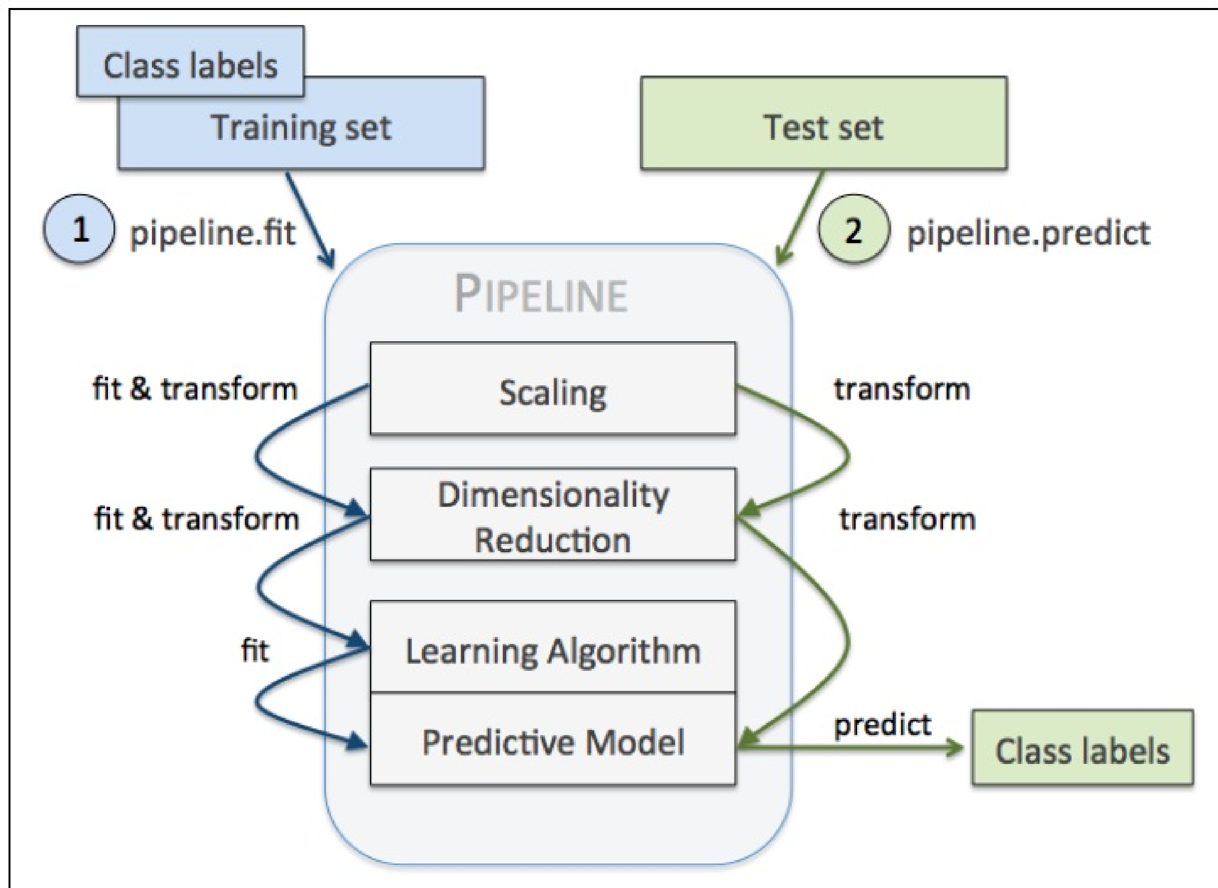
         Test Accuracy: 0.947
```

Pipeline对象接收元组构成的列表作为输入，每个元组第一个值作为变量名，元组第二个元素是sklearn中的transformer或Estimator。

管道中间每一步由sklearn中的transformer构成，最后一步是一个Estimator。我们的例子中，管道包含两个中间步骤，一个StandardScaler和一个PCA，这俩都是transformer，逻辑斯蒂回归分类器是Estimator。

当管道pipe_lr执行fit方法时，首先StandardScaler执行fit和transform方法，然后将转换后的数据输入给PCA，PCA同样执行fit和transform方法，最后将数据输入给LogisticRegression，训练一个LR模型。

对于管道来说，中间有多少个transformer都可以。管道的工作方式可以用下图来展示(一定要注意管道执行fit方法，而transformer要执行fit_transform)：


Class labels
Training set
Test set
1
pipeline.fit
2
pipeline.predict
PIPELINE
fit & transform
Scaling
transform
fit & transform
Dimensionality Reduction
transform
fit
Learning Algorithm
Predictive Model
predict
Class labels

K折交叉验证评估模型性能

训练机器学习模型的关键一步是要评估模型的泛化能力。如果我们训练好模型后，还是用训练集取评估模型的性能，这显然是不符合逻辑的。一个模型如果性能不好，要么是因为模型过于复杂导致过拟合(高方差)，要么是模型过于简单导致导致欠拟合(高偏差)。可是用什么方法评价模型的性能呢？这就是这一节要解决的问题，你会学习到两种交叉验证计数，holdout交叉验证和k折交叉验证， 来评估模型的泛化能力。

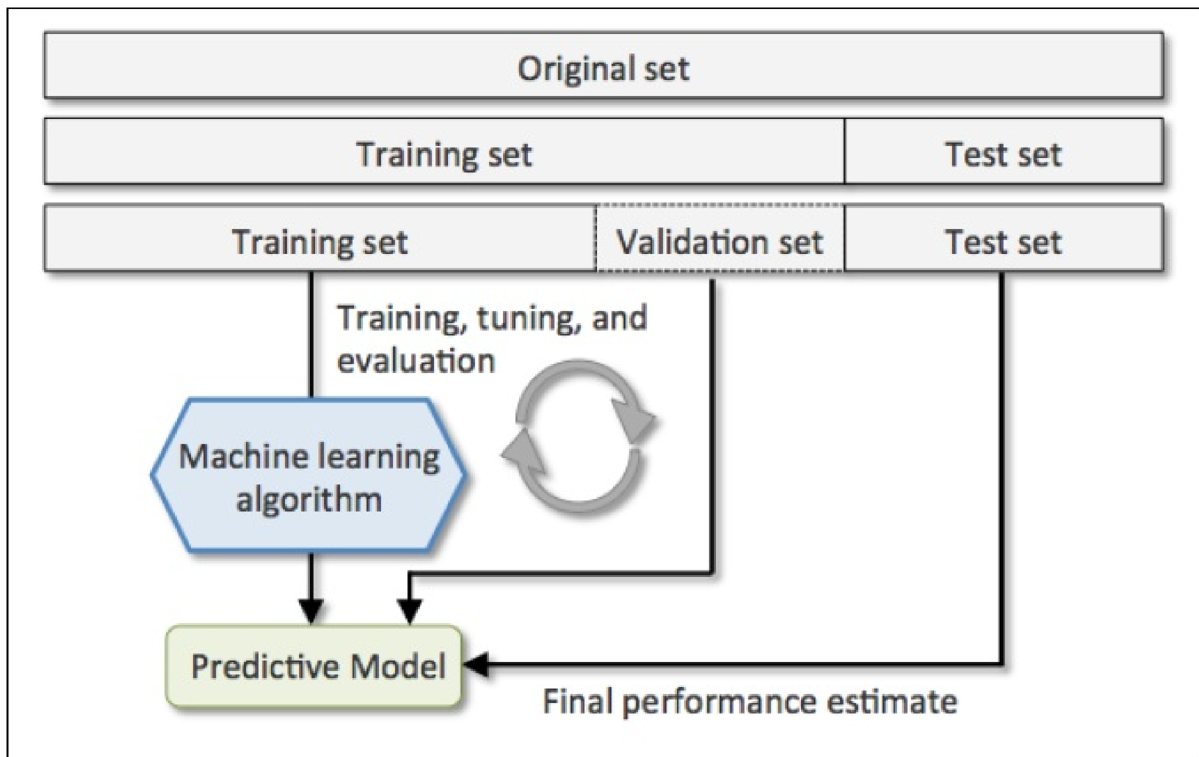
holdout method

评估模型泛化能力的典型方法是holdout交叉验证(holdout cross validation)。holdout方法很简单，我们只需要将原始数据集分割为训练集和测试集，前者用于训练模型，后者用于评估模型的性能。

不过，在训练模型这一步，我们非常关心如何选择参数来提高模型的预测能力，而选择参数这一步被称为模型选择(model selection，译者注：不少资料将选择何种模型算法称为模型选择)，参数选择是非常重要的，因为对于同一种机器学习算法，如果选择不同的参数(超参数)，模型的性能会有很大差别。

如果在模型选择的过程中，我们始终用测试集来评价模型性能，这实际上也将测试集变相地转为了训练集，这时候选择的最优模型很可能是过拟合的。

更好的holdout方法是将原始训练集分为三部分：训练集、验证集和测试集。训练机用于训练不同的模型，验证集用于模型选择。而测试集由于在训练模型和模型选择这两步都没有用到，对于模型来说是未知数据，因此可以用于评估模型的泛化能力。下图展示了holdout方法的步骤：



当然holdout方法也有明显的缺点，它对数据分割的方式很敏感，如果原始数据集分割不当，这包括训练集、验证集和测试集的样本数比例，以及分割后数据的分布情况是否和原始数据集分布情况相同等等。所以，不同的分割方式可能得到不同的最优模型参数。

下一节，我们会学习到一种鲁棒性更好的模型评估方法，k折交叉沿则，即重复k次holdout方法提高鲁棒性。

k折交叉验证

k折交叉验证的过程，第一步我们使用不重复抽样将原始数据随机分为k份，第二步 k-1份数据用于模型训练，剩下那一份数据用于测试模型。然后重复第二步k次，我们就得到了k个模型和他的评估结果(译者注：为了减小由于数据分割引入的误差，通常k折交叉验证要随机使用不同的划分方法重复p次，常见的有10次10折交叉验证)。

然后我们计算k折交叉验证结果的平均值作为参数/模型的性能评估。使用k折交叉验证来寻找最优参数要比holdout方法更稳定。一旦我们找到最优参数，要使用这组参数在原始数据集上训练模型作为最终的模型。

k折交叉验证使用不重复采样，优点是每个样本只会在训练集或测试中出现一次，这样得到的模型评估结果有更低的方法。

下图演示了10折交叉验证：



至于k折中的k到底设定为多少，这个又是一个调参的过程，当然了，这一步很少有人会调参，一般都是用10.但是如果你的数据集特别小，我们当然希望训练集大一点，这时候就要设定大一点的k值，因为k越大，训练集在整个原始训练集的占比就越多。但是呢，k也不能太大，一则是导致训练模型个数过多，二则是k很大的情况下，各个训练集相差不多，导致高方差。要是你的数据集很大，那就把k设定的小一点咯，比如5.

其实呢，在将原始数据集划分为k部分的过程中，有很多不同的采样方法，比如针对非平衡数据的分层采样。分层采样就是在每一份子集中都保持原始数据集的类别比例。比如原始数据集正类：负类=3:1，这个比例也要保持在各个子集中才行。sklearn中实现了分层k折交叉验证哦：

```
In [16]: import numpy as np
```

```
In [17]: from sklearn.cross_validation import StratifiedKFold
```

```
In [18]: kfold = StratifiedKFold(y=y_train,
                                 n_folds=10,
                                 random_state=1)
```

```
In [19]: scores = []
```

```
In [21]: for k, (train, test) in enumerate(kfold):
             pipe_lr.fit(X_train[train], y_train[train])
             score = pipe_lr.score(X_train[test], y_train[test])
             scores.append(score)
             print('Fold: %s, Class dist.: %s, Acc: %.3f' % (k+1, np.bincount(y_train[train]), score))

         Fold: 1, Class dist.: [256 153], Acc: 0.891
         Fold: 2, Class dist.: [256 153], Acc: 0.978
         Fold: 3, Class dist.: [256 153], Acc: 0.978
         Fold: 4, Class dist.: [256 153], Acc: 0.913
         Fold: 5, Class dist.: [256 153], Acc: 0.935
         Fold: 6, Class dist.: [257 153], Acc: 0.978
         Fold: 7, Class dist.: [257 153], Acc: 0.933
         Fold: 8, Class dist.: [257 153], Acc: 0.956
         Fold: 9, Class dist.: [257 153], Acc: 0.978
         Fold: 10, Class dist.: [257 153], Acc: 0.956
```

```
In [22]: print('CV accuracy: %.3f +/- %.3f' % (np.mean(scores), np.std(scores)))

         CV accuracy: 0.950 +/- 0.029
```

sklearn就是这么方便，此外，更人性化的是sklearn还实现了一个直接得到交叉验证评估结果的方法cross_val_score(内部同样是分层k折交叉验证):

```
In [23]: from sklearn.cross_validation import cross_val_score
```

```
In [24]: scroes = cross_val_score(estimator=pipe_lr,
                                  X=X_train,
                                  y=y_train,
                                  cv=10,
                                  n_jobs=1)
```

```
In [25]: print('CV accuracy scores: %s' %scores)

         CV accuracy scores: [0.89130434782608692, 0.97826086956521741, 0.97826086956521741, 0.91304347826086951, 0.93478260869565222, 0.977777777777
         77775, 0.93333333333333335, 0.9555555555555556, 0.97777777777777775, 0.9555555555555556]
```

```
In [27]: print('CV accuracy: %.3f +/- %.3f' %(np.mean(scores), np.std(scores)))

         CV accuracy: 0.950 +/- 0.029
```

cross_val_score方法还支持并行计算。

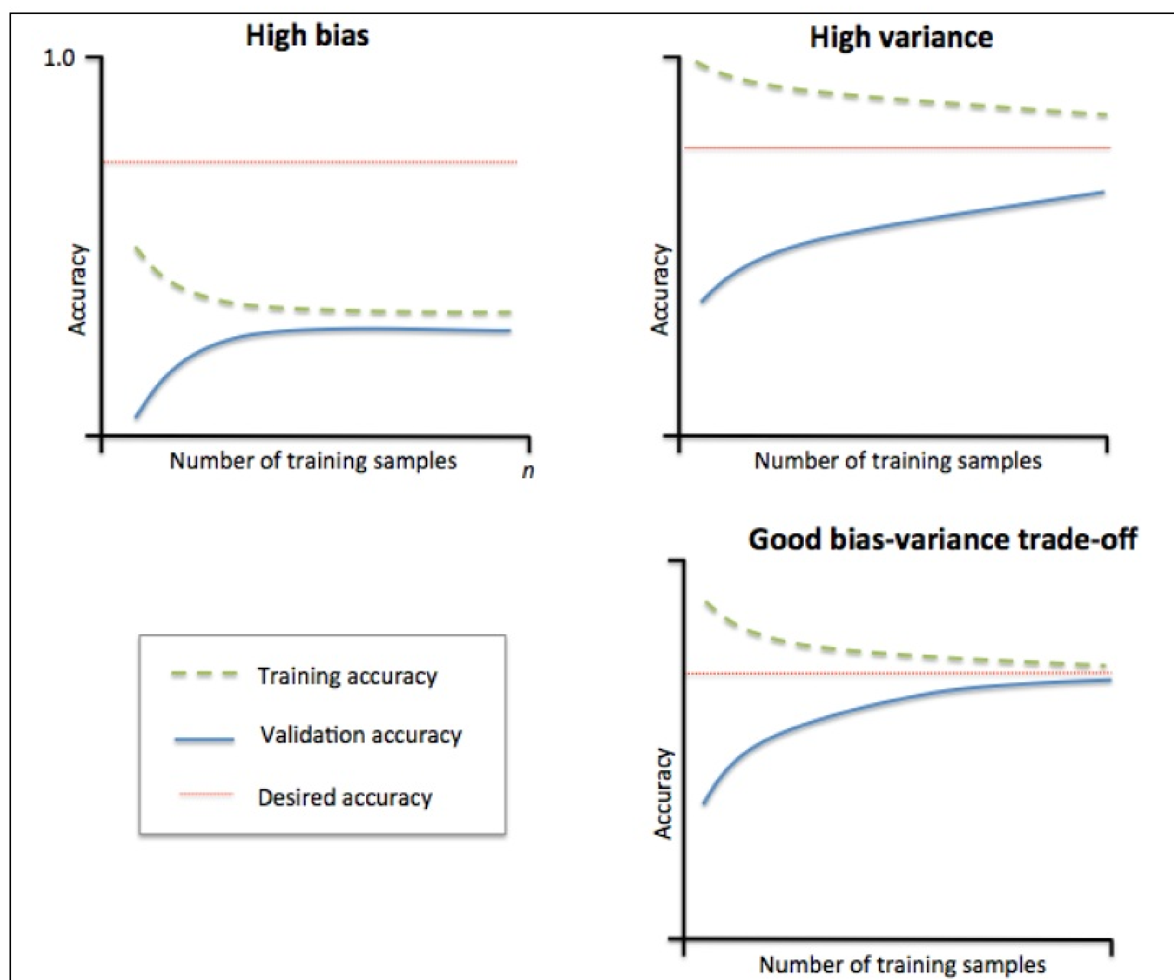
Note 交叉验证是如何评估泛化能力的方差，这个问题超出了本书的范围，如果你感兴趣，可以阅读 *Analysis of Variance of Cross-validation Estimators of the Generalized Error. Journal of Machine Learning Research, 6:1127-1168, 2005*

使用学习曲线和验证曲线 调试算法

这一节我们学习两个非常有用的诊断方法，可以用来提高算法的表现。他们就是学习曲线(learning curve)和验证曲线(validation curve)。学习曲线可以判断学习算法是否过拟合或者欠拟合。

使用学习曲线判别偏差和方差问题

如果一个模型相对于训练集来说过于复杂，比如参数太多，则模型很可能过拟合。避免过拟合的手段包含增大训练集，但这是不容易做到的。通过画出不同训练集大小对应的训练集和验证集准确率，我们能够很轻松滴检测模型是否方差偏高或偏差过高，以及增大训练集是否有用。



上图的左上角子图中模型偏差很高。它的训练集和验证集准确率都很低，很可能是欠拟合。解决欠拟合的方法就是增加模型参数，比如，构建更多的特征，减小正则项。

上图右上角子图中模型方差很高，表现就是训练集和验证集准确率相差太多。解决过拟合的方法有增大训练集或者降低模型复杂度，比如增大正则项，或者通过特征选择减少特征数。

这俩问题可以通过验证曲线解决。

我们先看看学习曲线是怎么回事吧：

In [28]:
```
import matplotlib.pyplot as plt
```

In [29]:
```
from sklearn.learning_curve import learning_curve
```

In [30]:
```
pipe_lr = Pipeline([
        ('scl', StandardScaler()),
        ('clf', LogisticRegression(
        penalty='l2', random_state=0))])
```

In [31]:
```
train_sizes, train_scores, test_scores = learning_curve(estimator=pipe_lr,
                                                        X=X_train,
                                                        y=y_train,
                                                        train_sizes=np.linspace(0.1, 1.0, 10),
                                                        cv=10,
                                                        n_jobs=1)
```

In [32]:
```
train_mean = np.mean(train_scores, axis=1)
```

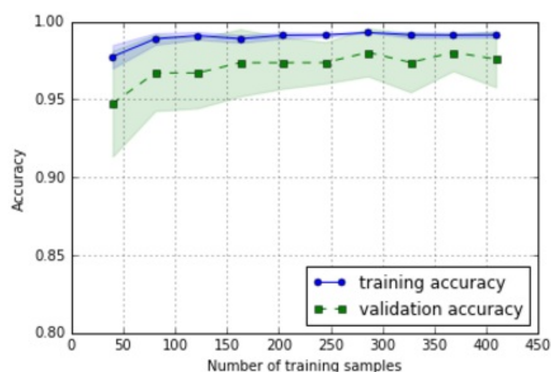
In [33]:
```
train_std = np.std(train_scores, axis=1)
```

In [34]:
```
test_mean = np.mean(test_scores, axis=1)
```

In [35]:
```
test_std = np.std(test_scores, axis=1)
```

In [37]:
```
plt.plot(train_sizes, train_mean,
         color='blue', marker='o',
         markersize=5,
         label='training accuracy')
plt.fill_between(train_sizes,
                 train_mean + train_std,
                 train_mean - train_std,
                 alpha=0.15, color='blue')
plt.plot(train_sizes, test_mean,
         color='green', linestyle='--',
         marker='s', markersize=5,
         label='validation accuracy')
plt.fill_between(train_sizes,
                 test_mean + test_std,
                 test_mean - test_std,
                 alpha=0.15, color='green')

plt.grid()
plt.xlabel('Number of training samples')
plt.ylabel('Accuracy')
plt.legend(loc='lower right')
plt.ylim([0.8, 1.0])
plt.show()
```



learning_curve中的train_sizes参数控制产生学习曲线的训练样本的绝对/相对数量，此处，我们设置的train_sizes=np.linspace(0.1, 1.0, 10)，将训练集大小划分为10个相等的区间。learning_curve默认使用分层k折交叉验证计算交叉验证的准确率，我们通过cv设置k。

上图中可以看到，模型在测试集表现很好，不过训练集和测试集的准确率还是有一段小间隔，可能是模型有点过拟合。

用验证曲线解决过拟合和欠拟合

验证曲线是非常有用的工具，他可以用来提高模型的性能，原因是他能处理过拟合和欠拟合问题。

验证曲线和学习曲线很相近，不同的是这里画出的是不同参数下模型的准确率而不是不同训练集大小下的准确率：

In [44]:
```
from sklearn.learning_curve import validation_curve
```

In [45]:
```
param_range = [0.001, 0.01, 0.1, 1.0, 10.0, 100.0]
```

In [47]:
```
train_scores, test_scores = validation_curve(estimator=pipe_lr,
                                             X=X_train,
                                             y=y_train,
                                             param_name='clf__C',
                                             param_range=param_range,
                                             cv=10)
```

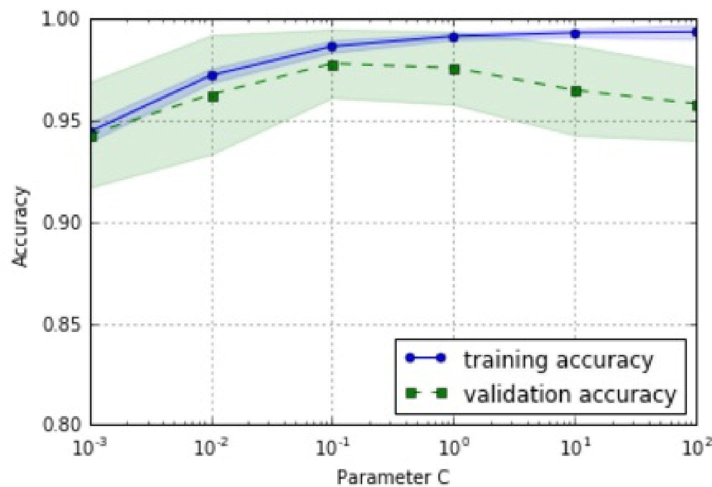
In [48]:
```
train_mean = np.mean(train_scores, axis=1)
```

In [49]:
```
train_std = np.std(train_scores, axis=1)
```

In [50]:
```
test_mean = np.mean(test_scores, axis=1)
```

In [51]:
```
test_std=np.std(test_scores, axis=1)
```

```
In  [52]: plt.plot(param_range, train_mean,
              color='blue', marker='o',
              markersize=5,
              label='training accuracy')
          plt.fill_between(param_range, train_mean + train_std,
              train_mean - train_std, alpha=0.15,
              color='blue')
          plt.plot(param_range, test_mean,
              color='green', linestyle='--',
              marker='s', markersize=5,
              label='validation accuracy')
          plt.fill_between(param_range,
              test_mean + test_std,
              test_mean - test_std,
              alpha=0.15, color='green')
          plt.grid()
          plt.xscale('log')
          plt.legend(loc='lower right')
          plt.xlabel('Parameter C')
          plt.ylabel('Accuracy')
          plt.ylim([0.8, 1.0])
          plt.show()
```



我们得到了参数C的验证曲线。

和learning_curve方法很像，validation_curve方法使用采样k折交叉验证来评估模型的性能。在validation_curve内部，我们设定了用来评估的参数，这里是C,也就是LR的正则系数的倒数。

观察上图，最好的C值是0.1。

通过网格搜索调参

机器学习算法中有两类参数：从训练集中学习到的参数，比如逻辑斯蒂回归中的权重参数，另一类是模型的超参数，也就是需要人工设定的参数，比如正则项系数或者决策树的深度。

前一节，我们使用验证曲线来提高模型的性能，实际上就是找最优参数。这一节我们学习另一种常用的超参数寻优算法：网格搜索(grid search)。

网格搜索听起来高大上，实际上简单的一笔，就是暴力搜索而已，我们事先为每个参数设定一组值，然后穷举各种参数组合，找到最好的那一组。

In [53]:
```
from sklearn.grid_search import GridSearchCV
from sklearn.svm import SVC
```

In [54]:
```
pipe_svc = Pipeline([('scl', StandardScaler()),
                     ('clf', SVC(random_state=1))])
```

In [55]:
```
param_range = [0.0001, 0.001, 0.01, 0.1, 1.0, 10.0, 100.0, 1000.0]
```

In [56]:
```
param_grid = [{
        'clf__C': param_range,
        'clf__kernel':['linear']},
              {
        'clf__C': param_range,
        'clf__gamma': param_range,
        'clf__kernel': ['rbf']}]
```

In [57]:
```
gs = GridSearchCV(estimator=pipe_svc,
                  param_grid = param_grid,
                  scoring='accuracy',
                  cv=10,
                  n_jobs=-1)
```

In [58]:
```
gs = gs.fit(X_train, y_train)
```

In [59]:
```
print(gs.best_score_)
```

```
0.978021978022
```

In [60]:
```
print(gs.best_params_)
```

```
{'clf__C': 0.1, 'clf__kernel': 'linear'}
```

GridSearchCV中param_grid参数是字典构成的列表。对于线性SVM，我们只评估参数C；对于RBF核SVM，我们评估C和gamma。

最后， 我们通过best_parmas_得到最优参数组合。

sklearn人性化的一点是，我们可以直接利用最优参数建模(best_estimator_)：

```
In [61]: clf = gs.best_estimator_
```

```
In [62]: clf.fit(X_train, y_train)
```

```
Out[62]: Pipeline(steps=[('scl', StandardScaler(copy=True, with_mean=True, with_std=True)), ('clf', SVC(C=0.1, cache_size=200, class_weight=None, coe
f0=0.0,
  decision_function_shape=None, degree=3, gamma='auto', kernel='linear',
  max_iter=-1, probability=False, random_state=1, shrinking=True,
  tol=0.001, verbose=False))])
```

```
In [63]: print('Test accuracy: %.3f' % clf.score(X_test, y_test))
```

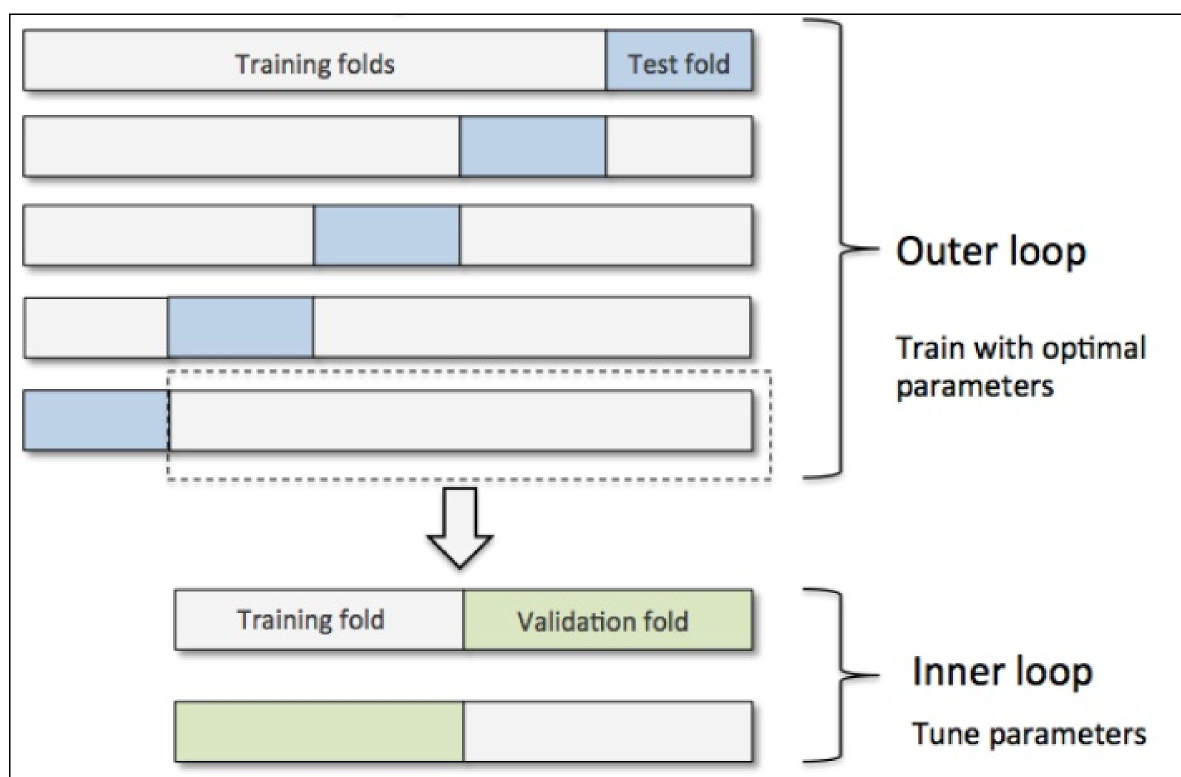
```
Test accuracy: 0.965
```

Note 网格搜索虽然不错，但是穷举过于耗时，sklearn中还实现了随机搜索，使用RandomizedSearchCV类，随机采样出不同的参数组合。

通过嵌套交叉验证选择算法

结合k折交叉验证和网格搜索是调参的好手段。可是如果我们想从茫茫算法中选择最合适的算法，用什么方法呢？这就是本节要介绍的嵌套交叉验证(nested cross validation)。 Varma和Simon在论文*Bias in Error Estimation When Using Cross-validation for Model Selection*中指出使用嵌套交叉验证得到的测试集误差几乎就是真实误差。

嵌套交叉验证外层有一个k折交叉验证将数据分为训练集和测试集。还有一个内部交叉验证用于选择模型算法。下图演示了一个5折外层交叉沿则和2折内部交叉验证组成的嵌套交叉验证，也被称为5*2交叉验证：



sklearn中可以如下使用嵌套交叉验证：

```
In [64]: gs = GridSearchCV(estimator=pipe_svc,
                          param_grid=param_grid,
                          scoring='accuracy',
                          cv=10,
                          n_jobs=-1)
```

```
In [65]: scores = cross_val_score(gs, X, y, scoring='accuracy', cv=5)
```

```
In [66]: print('CV accuracy: %.3f +/- %.3f' % (
             np.mean(scores), np.std(scores)))

         CV accuracy: 0.972 +/- 0.012
```

我们使用嵌套交叉验证比较SVm和决策树分类器：

In [67]:
```
from sklearn.tree import DecisionTreeClassifier
gs = GridSearchCV(
    estimator=DecisionTreeClassifier(random_state=0),
    param_grid=[
    {'max_depth': [1, 2, 3, 4, 5, 6, 7, None]}],
    scoring='accuracy',
    cv=5)
scores = cross_val_score(gs,
    X_train,
    y_train,
    scoring='accuracy',
    cv=5)
print('CV accuracy: %.3f +/- %.3f' % (
    np.mean(scores), np.std(scores)))
```

```
CV accuracy: 0.908 +/- 0.045
```

不同的性能评价指标

在前面几个章节，我们一直使用准确率(accuracy)来评价模型的性能，通常这是一个不错的选择。除此之外，还有不少评价指标哦，比如查准率(precision)、查全率(recall)和F1值(F1-score).

混淆矩阵

在讲解不同的评价指标之前，我们先来学习一个概念：混淆矩阵(confusion matrix), 能够展示学习算法表现的矩阵。混淆矩阵是一个平方矩阵，其中记录了一个分类器的TP(true positive)、TN(true negative)、FP(false positive)和FN(false negative):

		Predicted class	
		P	N
Actual Class	P	True Positives (TP)	False Negatives (FN)
	N	False Positives (FP)	True Negatives (TN)

计算着四个指标并不复杂，不过能不手算当然就不手算啦，sklearn中提供了confusion_matrix函数：

第七章 集成学习

前一章我们主要学习了怎样调参以及对模型进行评估。这一章，我们就要实际运用这些技巧，继而探索构建集成分类器的不同方法，集成分类器得到的结果通常比单个分类器要好。

这一章你将要学习：

- 基于投票的预测
- 通过可重复采用构建训练集，降低过拟合
- 以弱学习器为基础，从错误中学习来构建强学习器

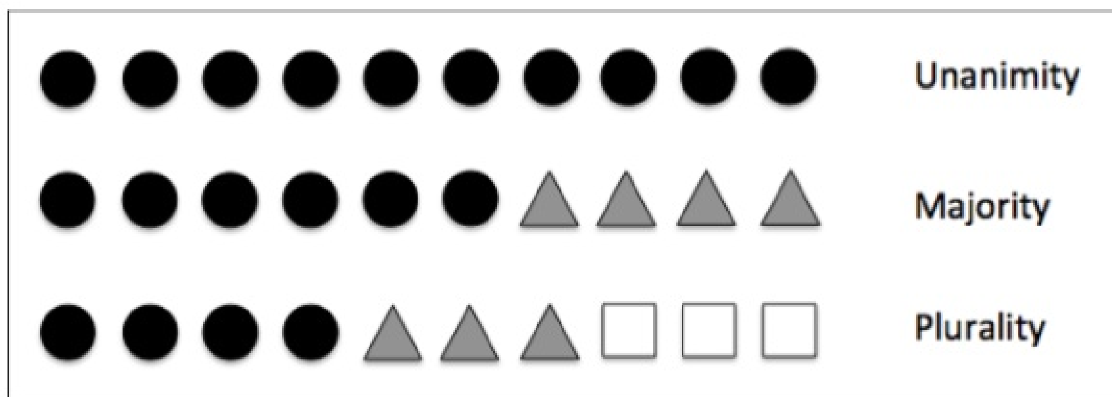
集成学习

集成学习背后的思想是将不同的分类器进行组合得到一个元分类器，这个元分类器相对于单个分类器拥有更好的泛化性能。比如，假设我们从10位专家那里分别得到了对于某个事件的预测结果，集成学习能够对这10个预测结果进行组合，得到一个更准确的预测结果。

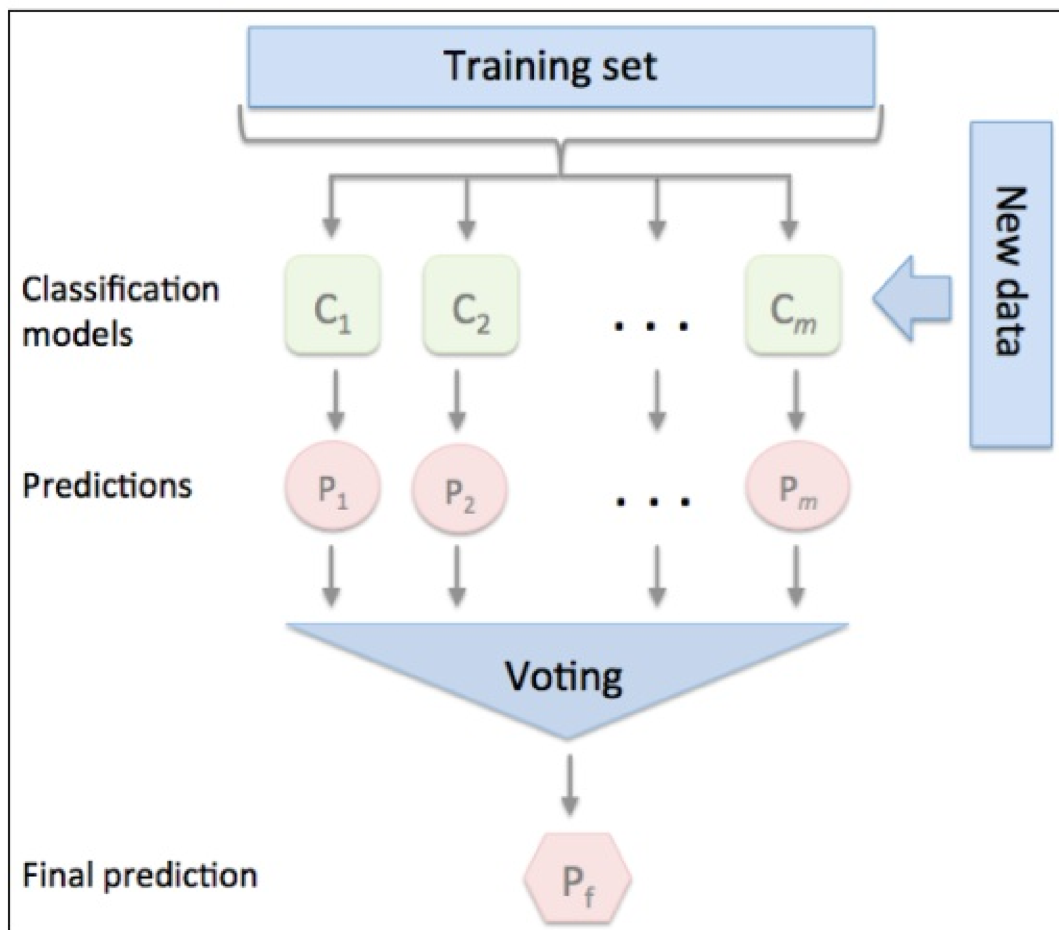
后面我们会学到，有不同的方法来创建集成模型，这一节我们先解决一个基本的问题：为什么要用集成学习？她为什么就比单个模型效果要好呢？

本书是为初学者打造的，所以集成学习这里我们也只关注最基本的集成方法：投票法(majority voting)。投票法意味着我们在得到最后的预测类别时，看看哪个类别是大多数单分类器都预测的，这里的大多数一般是大于50%。更严格来说，投票法只适用于二分类，当然他很容易就扩展到多分类情况: 多数表决(plurality voting).

下图展示了一个投票法的例子，一共10个基本分类器：



我们先用训练集训练m个不同的分类器$(C_1, \ldots, C_m)$, 这里的分类器可以是决策树、SVM或者LR等。我们当然也可以用同一种分类器，只不过在训练每一个模型时用不同的参数或者不同的训练集(比如自主采样法)。随机森林就是一个采用这种策略的例子，它由不同的决策树模型构成。这图展示了用投票策略的集成方法步骤：



投票策略非常简单，我们收集每个单分类器c_j的预测类别 ,将票数最多的 作为预测结果：

$$\hat{y} = mode\{C_1(\boldsymbol{x}), C_2(\boldsymbol{x}), \ldots, C_m(\boldsymbol{x})\}$$

以二分类为例，类别class1=-1, class2=+1, 投票预测的过程如下, 把每个单分类器的预测结果相加，如果值大于0，预测结果为正类，否则为负类：

$$C(\boldsymbol{x}) = sign\left[\sum_j^m C_j(\boldsymbol{x})\right] = \begin{cases} 1 & if \sum_i C_j(\boldsymbol{x}) \geq 0 \\ -1 & otherwise \end{cases}$$

读到这里，我想大家都有一个疑问：凭啥集成学习就比单分类器效果好？道理很简单(一点点组合数学知识)，假设对于一个二分类问题，有n个单分类器，每个单分类器有相等的错误率 ，并且单分类器之间相互独立，错误率也不相关。 有了这些假设，我们可以计算集成模型的错误概率：

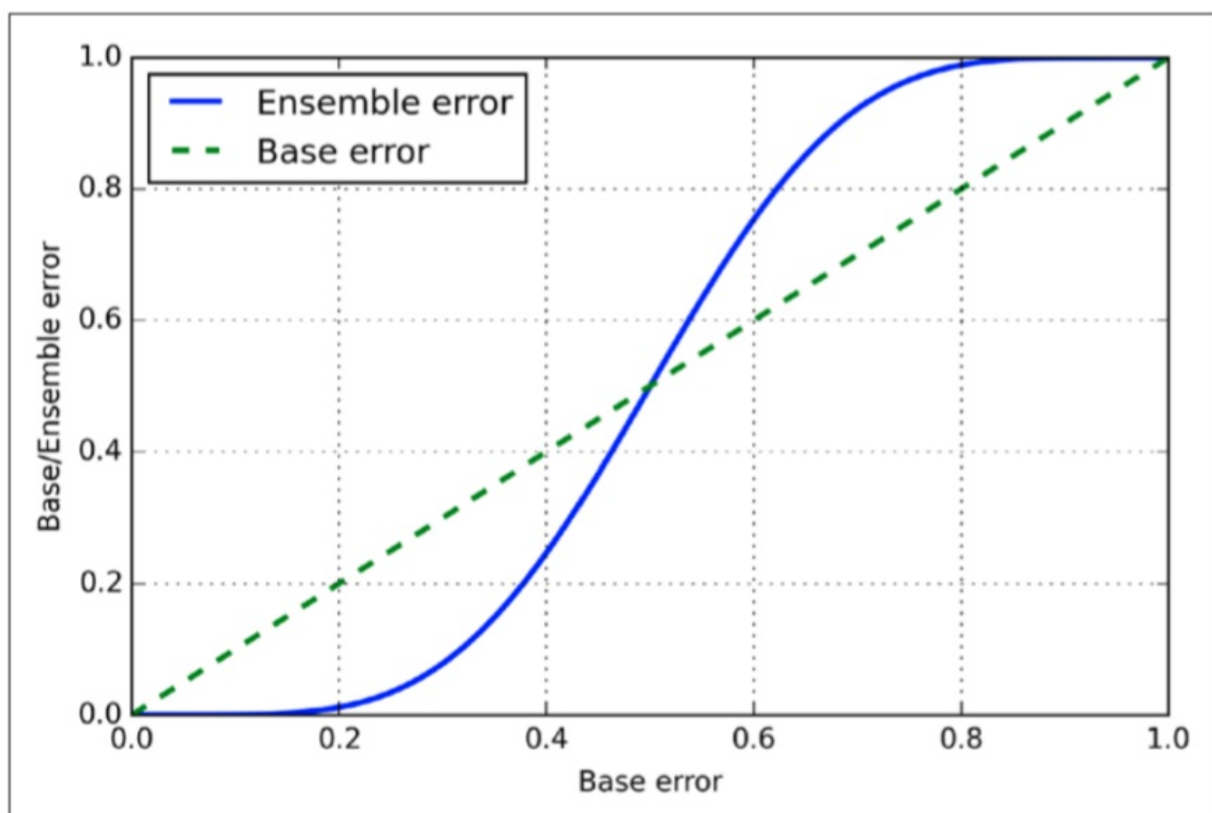
$$P(y \geq k)=\sum_{k}^{n}\left\langle\begin{matrix} n \\ k \end{matrix}\right\rangle \varepsilon^{k}(1-\varepsilon)^{n-k}=\varepsilon_{ensemble}$$

如果n=11，错误率为0.25，要想集成结果预测错误，至少要有6个单分类器预测结果不正确，错误概率是：

$$P(y \geq k)=\sum_{k=6}^{n}\left\langle\begin{matrix} 11 \\ k \end{matrix}\right\rangle 0.25^{k}(1-\varepsilon)^{11-k}=0.034$$

集成结果错误率才0.034哦，比0.25小太多。继承结果比单分类器好，也是有前提的，就是你这个单分类器的能力不能太差，至少要比随机猜测的结果好一点，至少。

从下图可以看出，只要单分类器的表现不太差，集成学习的结果总是要好于单分类器的。



结合不同的分类算法进行投票

这一节学习使用sklearn进行投票分类，看一个具体的例子，数据集采用Iris数据集，只使用sepal width和petal length两个维度特征，类别我们也只是用两类：Iris-Versicolor和Iris-Virginica，评判标准使用ROC AUC。

In [1]:
```
from sklearn import datasets
from sklearn.cross_validation import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import LabelEncoder
```

In [2]:
```
iris = datasets.load_iris()
X, y = iris.data[50:, [1, 2]], iris.target[50:]
```

In [3]:
```
le = LabelEncoder()
```

In [4]:
```
y = le.fit_transform(y)
```

我们将数据集平均分为训练集和测试集。

In [5]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5, random_state=1)
```

我们训练三个不同的分类器: LR、决策树和KNN。

In [6]:
```
from sklearn.cross_validation import cross_val_score
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.pipeline import Pipeline
import numpy as np
```

In [7]:
```
clf1 = LogisticRegression(penalty='l2', C=0.001, random_state=0)
```

In [9]:
```
clf2 = DecisionTreeClassifier(max_depth=1, criterion='entropy', random_state=0)
```

In [10]:
```
clf3 = KNeighborsClassifier(n_neighbors=1, p=2, metric='minkowski')
```

In [11]:
```
pipe1 = Pipeline([['sc', StandardScaler()],
                  ['clf', clf1]])
```

In [12]:
```
pipe2 = Pipeline([['sc', StandardScaler()],
                  ['clf', clf2]])
```

In [13]:
```
pipe3 = Pipeline([['sc', StandardScaler()],
                  ['clf', clf3]])
```

In [14]:
```
clf_labels = ['LR', 'DecisionTree', 'KNN']
```

In [17]:
```
print("10-fold cross validation:\n")
for clf, label in zip([pipe1, pipe2, pipe3], clf_labels):
    scores = cross_val_score(estimator=clf, X=X_train, y=y_train, cv=10, scoring='roc_auc')
    print("ROC AUC: %0.2f (+/- %0.2f) [%s]"
          % (scores.mean(), scores.std(), label))
```

```
10-fold cross validation:

ROC AUC: 0.92 (+/- 0.20) [LR]
ROC AUC: 0.92 (+/- 0.15) [DecisionTree]
ROC AUC: 0.93 (+/- 0.10) [KNN]
```

深度学习

深度学习作为当下最火热的机器学习子领域，本章及后续章节会对其进行简要介绍。

如果你之前从未了解过深度学习，建议你先阅读"neural networks and deep learning"或者本人之前的阅读笔记神经网络和深度学习。

本章并不会对神经网络进行理论介绍，而是准备从实践出发，直接介绍目前最好用的深度学习框架（之一）PyTorch。

PyTorch出自Facebook，以Python为第一语言，不论是文档还是API，都很容易上手。

这里记录一下阅读PyTorch Tutorial时的笔记。

原文链接